FMCAD 2013

Formal Methods in Computer–Aided Design

Portland, OR, USA, 20–23 October 2013

Edited by Barbara Jobstmann and Sandip Ray

In cooperation with
ACM Special Interest Group on Programming Languages
ACM Special Interest Group on Software Engineering

Technical co-sponshorship of IEEE Council on Electronic Design Automation

Table of Contents

Preface

The International Conference on Formal Methods in Computer-Aided Design, FMCAD, is a series of conferences on the theory and application of formal methods to the computer-aided design and verification of hardware and systems. The thirteenth conference in the series, FMCAD 2013, was held in Portland, OR, USA, October 20-23.

In the past, FMCAD took place in the United States on even years and its sister conference CHARME was held in Europe on odd years. In 2006, these two conferences merged to form an annual conference with a unified international community. The merged conference inherited the name FMCAD, and is now held annually. 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 the spectrum of formal aspects of computer-aided system design, including verification, specification, synthesis, and testing. This year, the conference received in-cooperation status with ACM under the Special Interest Group on Programming Languages and the Special Interest Group on Software Engineering. It also received technical sponsorship from the IEEE Council on Electronic Design Automation. Three additional events were co-located with the conference this year: (1) MEMOCODE 2013, the Eleventh ACM-IEEE International Conference on Formal Methods and Models of Codesign, (2) DIFTS 2013, Workshop on Design and Implementation of Formal Tools and Systems, and (3) the Hardware Model Checking Competition (HWMCC). The 2013 FMCAD Proceedings are anticipated to be available through the ACM Digital Library, the IEEE Xplore Digital Library, or as a free download from the FMCAD Website.

FMCAD 2013 received 79 submissions (after discounting for withdrawn submissions). Each submission was reviewed by at least four reviewers, and some submissions received five or six reviews. After a long decision process that involved often vigorous discussions by Program Committee members and subreviewers, 30 submissions were eventually selected for presentation at the conference — 23 as regular papers and 7 as short papers. The accepted papers covered topics ranging from model checking and solver technology to design for verification. Moreover, they addressed a broad spectrum of abstraction levels, including analog and mixed-signal, low-level software/hardware interfaces, networking systems, and synchronous hardware designs and software programs.

A novelty of this year's conference is the Student Forum, intended specifically to attract students by providing them with a platform for introducing their research to the FMCAD community and obtain feedback. The forum included short presentations and a poster by a student author of each accepted submission.

Besides reviewed submissions, the program includes two keynote presentations, four tutorials, and a panel. Pranav Ashar, Chief Technology Officer of Real Intent, gave a keynote titled "Static Verification Based Signoff — A Key Enabler for Managing Verification Complexity in the Modern SoC". Lori Clarke, Professor at the University of Massachusetts, gave a keynote titled "Using Process Modeling and Analysis Techniques to Reduce Errors in Healthcare". The tutorials were hosted jointly by FMCAD and MEMOCODE. Rajeev Alur, Professor at University of Pennsylvania, gave a tutorial on Computer Augmented Program Engineering. Jim Grundy, Research Scientist at Intel Corporation, discussed challenges and opportunities in firmware validation. Somesh Jha and Tom Reps, Professors at the University of Wisconsin-Madison, together with their graduate student Bill Harris, spoke on security and techniques based on game-theory for enforcing security policies. Nate Foster, Assistant Professor at Cornell University and Arjun Guha, Assistant Professor at University of Massachusetts, Amherst, together with Mark Reitblatt, Ph.D. student at Cornell University, and Cole Schlesinger, Ph.D. Student at Princeton University, gave a tutorial on software-defined networking. The panel, moderated by Panagiotis Manolios, discussed and debated on the role and importance of formal methods in computer science education, and on approaches to integrate the subject in graduate and undergraduate curricula.

We sincerely thank our industrial sponsors for their financial support of FMCAD 2013: Atrenta, Galois, IBM Corporation, Intel Corporation, Jasper Design Automation, Mentor Graphics, Microsoft Corporation, NEC Labs America, NVIDIA, OneSpin Solutions, Oski Technology, Real Intent, Synopsys, and Xpliant. We thank FMCAD Inc. for continuous support of the conference series. We owe a large debt to this year's organizing committee, composed of Joe Leslie-Hurd (Local Arrangements), Julien Schmaltz (Publication), Chao Yan (Publicity), Thomas Wahl (Student Forum), Malay Ganai (Tutorials), and Shilpi Goel (Webmaster). Special thanks are due to Vigyan Singhal for his untiring efforts to secure industrial sponsorship on behalf of FMCAD. We also thank the members of the FMCAD Steering Committee Jason Baumgartner, Armin Biere, Aarti Gupta, Warren Hunt, and Panagiotis Manolios, for their kind advice during the conference preparation process. Big thanks to all members of the Program Committee who, with the help of many subreviewers, did a stellar job not only of selecting this year's exciting program, but also of providing feedback to the authors to help them improve their papers for publication. Last but not least, the conference would not be possible without all the authors who submitted papers and all the attendees.

Barbara Jobstmann and Sandip Ray
(FMCAD 2013 Co-Chairs)

Conference Organization

General and Program Co-chairs

Barbara Jobstmann, EPFL, Jasper Design Automation, and CNRS-Verimag
Sandip Ray, Intel Corporation

Local Arrangements Chair

Joe Leslie-Hurd, Intel Corporation

Publications Chair

Julien Schmaltz, Open University of The Netherlands

Tutorial Chair

Malay Ganai, NEC Laboratories America

Student Forum Chair

Thomas Wahl, Northeastern University

Publicity Chair

Chao Yan, Intel Corporation

Webmaster

Shilpi Goel, The University of Texas at Austin

Steering Committee

Jason Baumgartner, IBM
Armin Biere, Johannes Kepler University in Linz
Aarti Gupta, NEC Labs America
Warren A. Hunt, Jr., The University of Texas at Austin
Panagiotis Manolios, Northeastern University

Program Committee

Jason Baumgartner, IBM Corporation
Dirk Beyer, University of Passau
Armin Biere, Johannes Kepler University
Per Bjesse, Synopsys
Nikolaj Bjorner, Microsoft Research
Roderick Bloem, TU Graz
Gianpiero Cabodi, Politecnico di Torino
Hana Chockler, IBM Research
Alessandro Cimatti, FBK-irst
Koen Claessen, Chalmers University of Technology
Malay Ganai, NEC Labs America
Steven German, IBM
Ganesh Gopalakrishnan, University of Utah
Alberto Griggio, FBK-irst
Ziyad Hanna, Jasper Design Automation
Keijo Heljanko, Aalto University
Alan Hu, University of British Columbia

William Hung, Synopsys
Warren Hunt, University of Texas
Susmit Jha, Strategic CAD Lab, Intel
Shuvendu Lahiri, Microsoft Research
Panagiotis Manolios, Northeastern University
Tom Melham, University of Oxford
John O'Leary, Intel Corporation
Lee Pike, Galois
Ruzica Piskac, Yale University
Philipp Ruemmer, Uppsala University
Cesar Sanchez, IMDEA Software Institute
Julien Schmaltz, Open University of The Netherlands
Natasha Sharygina, University of Lugano
Anna Slobodova, Centaur Technology
Niklas Sorensson, Mentor Graphics
Daryl Stewart, ARM
Thomas Wahl, Northeastern University
Georg Weissenbacher, Vienna University of Technology

External Reviewers

Karam Abdelkader, Francesco Alberti, Gadi Aleksandrowicz, Leonardo Alt, Gogul Balakrishnan, Samuel Bayless, Aaron Bradley, Pavol Cerny, Harsh Raju Chamarthi, Xin Chen, Jason Dagit, Abhijit Davare, Iavor Diatchki, Bruno Dutertre, Rüdiger Ehlers, Pontus Ekberg, Emanuelle Encrenaz, Grigory Fedyukovich, Bernd Finkbeiner, Goran Frehse, Khalil Ghorbal, Shilpi Goel, Eugene Goldberg, Mark Greenstreet, Jim Grundy, Frédéric Haziza, Joe Hendrix, Marijn Heule, Pat Hickey, Håkan Hjort, Georg Hofferek, Andreas Holzer, Antti Hyvärinen, Alexander Ivrii, Swen Jacobs, Mitesh Jaiin, Himanshu Jain, Mitesh Jain, Sumit Kumar Jha, Kevin Jones, Sebastiaan Joosten, Kari Kähkönen, Roope Kaivola, Matt Kaufmann, Ayrat Khalimov, Zurab Khasidashvili, Johannes Kloos, Alfred Koelbl, Bettina Könighofer, Robert Könighofer, Laura Kovacs, Tuomas Kuismin, Armando Lezama, Wenchao Li, Scott Little, Peizun Liu, Stefan Löwe, Carmelo Loiacono, Oded Margalit, Johan Mårtensson, Sergio Mover, Filip Nikšić, Xiaoyue Pan, Vasilis Papavasileiou, Paolo Pasini, Flavio M. de Paula, Denis Poitrenaud, Corneliu Popeea, Sylvie Putot, Stefano Quer, Jaideep Ramachandran, John Regehr, Heinz Riener, Simone Fulvio Rollini, Sitvanit Ruah, Leonid Ryzhyk, Indranil Saha, Jun Sawada, Bas Schaafsma, Martina Seidl, Ben Selfridge, Antti Tapani Siirtola, Radu Siminiceanu, Rohit Sinha, Andreas Stahlbauer, Baruch Sterin, Pavle Subotic, Philippe Suter, Sol Swords, Aaron Tomb, Celina G. Val, Danilo Vendraminetto, Freek Verbeek, Yakir Vizel, Chao Wang, Markus Wedler, Philipp Wendler, Nathan Wetzler, Josef Widder, Siert Wieringa, Simon Winwood, Guy Wolfovitz, Jessie Xu, Karen Yorav, Andy Yu, Jun Yuan, Aleksandar Zeljić, Yan Zhang

Syntax-Guided Synthesis

Rajeev Alur[†] Rastislav Bodik[‡] Garvit Juniwal[‡] Milo M. K. Martin[†] Mukund Raghothaman[†]
Sanjit A. Seshia[‡] Rishabh Singh[♯] Armando Solar-Lezama[♯] Emina Torlak[‡] Abhishek Udupa[†]
[†]University of Pennsylvania [‡]University of California, Berkeley [♯]Massachusetts Institute of Technology

***Abstract*—The classical formulation of the program-synthesis problem is to find a program that meets a correctness specification given as a logical formula. Recent work on program synthesis and program optimization illustrates many potential benefits of allowing the user to supplement the logical specification with a syntactic template that constrains the space of allowed implementations. Our goal is to identify the core computational problem common to these proposals in a logical framework. The input to the *syntax-guided synthesis* problem (SyGuS) consists of a background theory, a semantic correctness specification for the desired program given by a logical formula, and a syntactic set of candidate implementations given by a grammar. The computational problem then is to find an implementation from the set of candidate expressions so that it satisfies the specification in the given theory. We describe three different instantiations of the *counter-example-guided-inductive-synthesis* (CEGIS) strategy for solving the synthesis problem, report on prototype implementations, and present experimental results on an initial set of benchmarks.**

I. Introduction

In *program verification*, we want to check if a program satisfies its logical specification. Contemporary verification tools vary widely in terms of source languages, verification methodology, and the degree of automation, but they all rely on repeatedly invoking an SMT (Satisfiability Modulo Theories) solver. An SMT solver determines the truth of a given logical formula built from typed variables, logical connectives, and typical operations such as arithmetic and array accesses (see [1], [2]). Despite the computational intractability of these problems, modern SMT solvers are capable of solving instances with thousands of variables due to sustained innovations in core algorithms, data structures, decision heuristics, and performance tuning by exploiting the architecture of contemporary processors. A key driving force for this progress has been the standardization of a common interchange format for benchmarks called SMT-LIB (see smt-lib.org) and the associated annual competition (see smtcomp.org). These efforts have proved to be instrumental in creating a virtuous feedback loop between developers and users of SMT solvers: with the availability of open-source and highly optimized solvers, researchers from verification and other application domains find it beneficial to translate their problems into the common format instead of attempting to develop their own customized tools from scratch, and the limitations of the current SMT tools are constantly exposed by the ever growing repository of different kinds of benchmarks, thereby spurring greater innovation for improving the solvers.

In *program synthesis*, we wish to automatically synthesize an implementation for the program that satisfies the given correctness specification. A mature synthesis technology has the potential of even greater impact on software quality than program verification. Classically, program synthesis is viewed as a problem in deductive theorem proving: a program is derived from the constructive proof of the theorem that states that for all inputs, there exists an output, such that the desired correctness specification holds (see [3]). Our work is motivated by a recent trend in synthesis in which the programmer, in addition to the correctness specification, provides a syntactic template for the desired program. For instance, in the programming approach advocated by the SKETCH system, a programmer writes a partial program with incomplete details, and the synthesizer fills in the missing details using user-specified assertions as the correctness specification [4]. We call such an approach to synthesis *syntax-guided synthesis* (SyGuS). Besides program sketching, a number of recent efforts such as synthesis of loop-free programs [5], synthesis of Excel macros from examples [6], program de-obfuscation [7], synthesis of protocols from the skeleton and example behaviors [8], synthesis of loop-bodies from pre/post conditions [9], integration of constraint solvers in programming environments for program completion [10], and super-optimization by finding equivalent shorter loop bodies [11], all are arguably instances of syntax-guided synthesis. Also related are techniques for automatic generation of invariants using templates and by learning [12]–[14], and recent work on solving quantified Horn clauses [15].

Existing formalization of the SMT problem and the interchange format does not provide a suitable abstraction for capturing the syntactic guidance. The computational engines used by the various synthesis projects mentioned above rely on a small set of algorithmic ideas, but have evolved independently with no mechanism for comparison, benchmarking, and sharing of back-ends. The main contribution of this paper is to define the *syntax-guided synthesis* (SyGuS) problem in a manner that (1) captures the computational essence of these recent proposals and (2) is based on more canonical formal frameworks such as logics and grammars instead of features of specific programming languages. In our formalization, the correctness specification of the function f to be synthesized is given as a logical formula φ that uses symbols from a background theory T. The syntactic space of possible implementations for f is described as a set L of expressions built from the theory T, and this set is specified using a grammar. The syntax-guided synthesis problem then is to find an implementation expression $e \in L$ such that the formula $\varphi[f/e]$ is valid in the theory T. To illustrate an application of the SyGuS-problem, suppose we want to find a completion of a partial program with holes so as to satisfy given assertions. A typical SyGuS-encoding of this task will translate the

ISBN 978-0-9835678-3-7/13.

concrete parts of the partial program and the assertions into the specification formula φ, while the holes will be represented with the unknown functions to be synthesized, and the space of expressions that can substitute the holes will be captured by the grammar.

Compared to the classical formulation of the synthesis problem that involves only the correctness specification, the syntax-guided version has many potential benefits. First, the user can use the candidate set L to limit the search-space for potential implementations, and this has significant computational benefits for solving the synthesis problem. Second, this approach gives the programmer the flexibility to express the desired artifact using a combination of syntactic and semantic constraints. Such forms of *multi-modal* specifications have the potential to make programming more intuitive. Third, the set L can be used to constrain the space of implementations for the purpose of performance optimizations. For example, to optimize the computation of the product of two two-by-two matrices, we can limit the search space to implementations that use only 7 multiplication operations, and such a restriction can be expressed only syntactically. Fourth, because the synthesis problem boils down to finding a correct expression from the syntactic space of expressions, this search problem lends itself to machine learning and inductive inference as discussed in Section III. Finally, it is worth noting that the statement "there exists an expression e in the language generated by a context-free grammar G such that the formula $\varphi[f/e]$ is valid in a theory T" cannot be translated to determining the truth of a formula in the theory T, even with additional quantifiers.

The rest of the paper is organized in the following manner. In Section II, we formalize the core problem of syntax-guided synthesis with examples. In Section III, we discuss a generic architecture for solving the proposed problem using the iterative *counter-example guided inductive synthesis* strategy [16] that combines a learning algorithm with a verification oracle. For the learning algorithm, we show how three techniques from recent literature can be adapted for our purpose: the enumerative technique generates the candidate expressions of increasing size relying on the input-output examples for pruning; the symbolic technique encodes parse trees of increasing size using variables and constraints, and it calls an SMT solver to find a parse tree consistent with all the examples encountered so far; and the stochastic search uniformly samples the set L of expressions as a starting point, and then executes (probabilistic) traversal of the graph where two expressions are neighbors if one can be obtained from the other by a single edit operation on the parse tree. We report on a prototype implementation of these three algorithms, and evaluate their performance on a number of benchmarks in Section IV.

II. Problem Formulation

At a high level, the functional synthesis problem consists of finding a function f such that some logical formula φ capturing the correctness of f is valid. In syntax-guided synthesis, the synthesis problem is constrained in three ways: (1) the logical symbols and their interpretation are restricted to a *background theory*, (2) the *specification* φ is limited to a first order formula in the background theory with all its variables universally quantified, and (3) the universe of possible functions f is restricted to syntactic expressions described by a *grammar*. We now elaborate on each of these points.

Background Theory: The syntax for writing specifications is the same as classical typed first-order logic, but the formulas are evaluated with respect to a specified background theory T. The theory gives the vocabulary used for constructing formulas, the set of values for each type, and the interpretation for each of the function and relation (predicate) symbols in the vocabulary. We are mainly interested in theories T for which well-understood decision procedures are available for determining satisfaction modulo T (see [1] for a survey). A typical example is the theory of *linear integer arithmetic* (LIA) where each variable is either a boolean or an integer, and the vocabulary consists of boolean and integer constants, standard boolean connectives, addition ($+$), comparison ($\leq$), and conditionals (*ITE*). Note that the background theory can be a combination of logical theories, for instance, LIA and the theory of uninterpreted functions with equality.

Correctness Specification: For the function f to be synthesized, we are given the type of f and a formula φ as its correctness specification. The formula φ is a Boolean combination of predicates from the background theory, involving universally quantified free variables, symbols from the background theory, and the function symbol f, all used in a type-consistent manner.

Example 1: Assuming the background theory is LIA, consider the specification of a function f of type $\mathit{int} \times \mathit{int} \mapsto \mathit{int}$:

$$\varphi_1 : \ f(x,y) = f(y,x) \ \wedge \ f(x,y) \geq x.$$

The free variables in the specification are assumed to be universally quantified: a given function f satisfies the above specification if the quantified formula $\forall x, y.\, \varphi_1$ holds, or equivalently, if the formula φ_1 is valid.

Set of Candidate Expressions: In order to make the synthesis problem tractable, the "syntax-guided" version allows the user to impose structural (syntactic) constraints on the set of possible functions f. The structural constraints are imposed by restricting f to the set L of functions defined by a given context-free grammar G_L. Each expression in L has the same type as that of the function f, and uses the symbols in the background theory T along with the variables corresponding to the formal parameters of f.

Example 2: Suppose the background theory is LIA, and the type of the function f is $\mathit{int} \times \mathit{int} \mapsto \mathit{int}$. We can restrict the set of expressions $f(x,y)$ to be linear expressions of the inputs by restricting the body of the function to expressions in the set L_1 described by the grammar below:

$$\mathit{LinExp} \ := \ x \mid y \mid \mathit{Const} \mid \mathit{LinExp} + \mathit{LinExp}$$

Alternatively, we can restrict $f(x,y)$ to conditional expressions with no addition by restricting the body terms from the set L_2 described by:

$$\begin{aligned} \mathit{Term} \ &:= \ x \mid y \mid \mathit{Const} \mid \mathit{ITE}(\mathit{Cond}, \mathit{Term}, \mathit{Term}) \\ \mathit{Cond} \ &:= \ \mathit{Term} \leq \mathit{Term} \mid \mathit{Cond} \wedge \mathit{Cond} \mid \neg \mathit{Cond} \mid (\mathit{Cond}) \end{aligned}$$

Grammars can be conveniently used to express a wide range of constraints, and in particular, to bound the depth and/or the size of the desired expression.

SyGuS Problem Definition: Informally, given the correctness specification φ and the set L of candidates, we want to find an expression $e \in L$ such that if we use e as an implementation of the function f, the specification φ is valid. Let us denote the result of replacing each occurrence of the function symbol f in φ with the expression e by $\varphi[f/e]$. Note that we need to take care of binding of input values during such a substitution: if f has two inputs that the expressions in L refer to by the variable names x and y, then the occurrence $f(e_1, e_2)$ in the formula φ must be replaced with the expression $e[x/e_1, y/e_2]$ obtained by replacing x and y in e by the expressions e_1 and e_2, respectively. Now we can define the *syntax-guided synthesis* problem, SyGuS for short, precisely:

> Given a background theory T, a typed function symbol f, a formula φ over the vocabulary of T along with f, and a set L of expressions over the vocabulary of T and of the same type as f, find an expression $e \in L$ such that the formula $\varphi[f/e]$ is valid modulo T.

Example 3: For the specification φ_1 presented earlier, if the set of allowed implementations is L_1 as shown before, there is no solution to the synthesis problem. On the other hand, if the set of allowed implementations is L_2, a possible solution is the conditional if-then-else expression $\mathit{ITE}(x \geq y, x, y)$.

In some special cases, it is possible to reduce the decision problem for syntax guided synthesis to the problem of deciding formulas in the background theory using additional quantification. For example, every expression in the set L_1 is equivalent to $ax+by+c$, for integer constants a, b, c. If φ is the correctness specification, then deciding whether there exists an implementation for f in the set L_1 corresponds to checking whether the formula $\exists\, a, b, c. \forall\, X. \varphi[f/ax + by + c]$ holds, where X is the set of all free variables in φ. This reduction was possible for L_1 because the set of all expressions in L_1 can be represented by a single parameterized expression in the original theory. However, the grammar may permit expressions of arbitrary depth which may not be representable in this way, as in the case of L_2.

Synthesis of Multiple Functions: A general synthesis problem can involve more than one unknown function. In principle, adding support for problems with more than one unknown function is merely a matter of syntactic sugar. For example, suppose we want to synthesize functions $f_1(x_1)$ and $f_2(x_2)$, with corresponding candidate expressions given by grammars G_1 and G_2, with start non-terminals S_1 and S_2, respectively. Both functions can be encoded with a single function $f_{12}(\mathit{id}, x_1, x_2)$. The set of candidate expressions is described by the grammar that contains the rules of G_1 and G_2 along with a new production $S := \mathit{ITE}(\mathit{id} = 0, S_1, S_2)$, with the new start non-terminal S. Then, every occurrence of $f_1(x_1)$ in the specification can be replaced with $f_{12}(0, x_1, *)$ and every call to $f_2(x_2)$ can be replaced with $f_{12}(1, *, x_2)$. Although adding support for multiple functions does not fundamentally increase the expressiveness of the notation, it does offer significant convenience in encoding real-world synthesis problems.

Let Expressions in Grammar Productions: The SMT-LIB interchange format for specifying constraints allows the use of let expressions as part of the formulas, and this is supported by our language also: $(\mathit{let}\ \ [\mathit{var} = e_1]\ \ e_2)$. While *let*-expressions in a specification can be desugared, the same does not hold when they are used in a grammar. As an example, consider the grammar below for the set of candidate expressions for the function $f(x, y)$:

$$\begin{array}{lcl} T & := & (\mathit{let}\ [z = U]\ z + z) \\ U & := & x \mid y \mid \mathit{Const} \mid U + U \mid U * U \mid (U) \end{array}$$

The top-level expression specified by this grammar is the sum of two identical subexpressions built using arithmetic operators, and such a structure cannot be specified using a standard context-free grammar. In the example above, every *let* introduced by the grammar uses the same variable name. If the application of *let*-expressions are nested in the derivation tree, the standard rules for shadowing of variable definitions determine which definition corresponds to which use of the variable.

SYNTH-LIB Input Format: To specify the input to the SyGuS problem, we have developed an interchange format, called SYNTH-LIB, based on the syntax of SMT-LIB2—the input format accepted by the SMT solvers (see smt-lib.org). The input for the SyGuS problem to synthesize the function f with the specification φ_1 in the theory LIA, with the grammar for the languages L_1 is encoded in SYNTH-LIB as:

```
(set-logic LIA)
(synth-fun f ((x Int) (y Int)) Int
  ((Start Int (x y
               (Constant Int)
               (+ Start Start)))))
(declare-var a Int)
(declare-var b Int)
(constraint (= (f a b) (f b a)))
(constraint (>= (f a b) a))
(check-synth)
```

Optimality Criterion: The answer to our synthesis problem need not be unique: there may be two expressions e_1 and e_2 in the set L of allowed expressions such that both implementations satisfy the correctness specification φ. Ideally, we would like to associate a cost with each expression, and consider the problem of *optimal synthesis* which requires the synthesis tool to return the expression with the least cost among the correct ones. A natural cost metric is the size of the expression. In presence of *let*-expressions, the size directly corresponds to the number of instructions in the corresponding straight-line code, and thus such a metric can be used effectively for applications such as super-optimization.

III. Inductive Synthesis

Algorithmic approaches to program synthesis range over a wide spectrum, from *deductive synthesis* to *inductive synthesis*. In deductive program synthesis (e.g., [3]), a program is synthesized by constructively proving a theorem, employing logical inference and constraint solving. On the other hand, inductive

synthesis [17]–[19] seeks to find a program matching a set of input-output examples. It is thus an instance of learning from examples, also termed as *inductive inference* or *machine learning* [20], [21]. Many current approaches to synthesis blend induction and deduction [22]; syntax guidance is usually a key ingredient in these approaches.

Inductive synthesizers generalize from examples by searching a restricted space of programs. In machine learning, this restricted space is called the *concept class*, and each element of that space is often called a candidate *concept*. The concept class is usually specified syntactically. Inductive learning is thus a natural fit for the syntax-guided synthesis problem introduced in this paper: the concept class is simply the set L of permissible expressions.

A. Synthesis via Active Learning

A common approach to inductive synthesis is to formulate the overall synthesis problem as one of *active learning* using a *query-based* model. Active learning is a special case of machine learning in which the learning algorithm can control the selection of examples that it generalizes from and can query one or more oracles to obtain both examples as well as labels for those examples. In our setting, we can consider the labels to be binary: positive or negative. A positive example is simply an interpretation to f in the background theory T that is consistent with the specification φ; i.e., it is a valuation to the arguments of the function symbol f along with the corresponding valuation of f that satisfies φ. A negative example is any interpretation of f that is not consistent with φ. We refer the reader to a paper by Angluin [23] for an overview of various models for query-based active learning.

In program synthesis via active learning, the query oracles are often implemented using deductive procedures such as model checkers or satisfiability solvers. Thus, the overall synthesis algorithm usually comprises a top-level inductive learning algorithm that invokes deductive procedures (query oracles); e.g., in our problem setting, it is intuitive, although not required, to implement an oracle using an SMT solver for the theory T. Even though this approach combines induction and deduction, it is usually referred to in the literature simply as "inductive synthesis." We will continue to use this terminology in the present paper.

Consider the syntax-guided synthesis problem of Sec. II. Given the tuple (T, f, φ, L), there are two important choices one must make to fix an inductive synthesis algorithm: (1) *search strategy:* How should one search the concept class L? and (2) *example selection strategy:* Which examples do we learn from?

B. Counterexample-Guided Inductive Synthesis

Counterexample-guided inductive synthesis (CEGIS) [16], [24] shown in Figure 1 is perhaps the most popular approach to inductive synthesis today. CEGIS has close connections to algorithmic debugging using counterexamples [19] and counterexample-guided abstraction refinement (CEGAR) [25]. This connection is no surprise, because both debugging and abstraction-refinement involve synthesis steps: synthesizing a repair in the former case, and synthesizing an abstraction function in the latter (see [22] for a more detailed discussion).

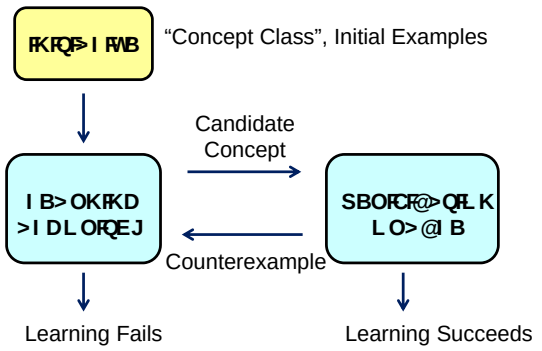


Fig. 1. Counterexample-Guided Inductive Synthesis (CEGIS)

The defining aspect of CEGIS is its example selection strategy: *learning from counterexamples provided by a verification oracle*. The learning algorithm, which is initialized with a particular choice of concept class L and possibly with an initial set of (positive) examples, proceeds by searching the space of candidate concepts for one that is consistent with the examples seen so far. There may be several such consistent concepts, and the search strategy determines the chosen candidate, an expression e. The concept e is then presented to the verification oracle $\mathcal{O}_V$, which checks the candidate against the correctness specification. $\mathcal{O}_V$ can be implemented as an SMT solver that checks whether $\varphi[f/e]$ is valid modulo the theory T. If the candidate is correct, the synthesizer terminates and outputs this candidate. Otherwise, the verification oracle generates a counterexample, an interpretation to the symbols and free variables in $\varphi[f/e]$ that falsifies it. This counterexample is returned to the learning algorithm, which adds the counterexample to its set of examples and repeats its search; note that the precise encoding of a counterexample and its use can vary depending on the details of the learning algorithm employed. It is possible that, after some number of iterations of this loop, the learning algorithm may be unable to find a candidate concept consistent with its current set of (positive/negative) examples, in which case the learning step, and hence the overall CEGIS procedure, fails.

Several search strategies are possible for learning a candidate expression in L, each with its pros and cons. In the following sections, we describe three different search strategies and illustrate the main ideas in each using a small example.

C. Illustrative Example

Consider the problem of synthesizing a program which returns the maximum of two integer inputs. The specification of the desired program $\max$ is given by:

$$\max(x, y) \geq x \ \wedge \ \max(x, y) \geq y \ \wedge \\ (\max(x, y) = x \vee \max(x, y) = y)$$

The search space is suitably defined by an expression grammar which includes addition, subtraction, comparison, conditional operators and the integer constants 0 and 1.

Expression to Verifier	Learned Test Input
x	$\langle x = 0, y = 1\rangle$
y	$\langle x = 1, y = 0\rangle$
1	$\langle x = 0, y = 0\rangle$
$x + y$	$\langle x = 1, y = 1\rangle$
$\mathit{ITE}(x \leq y, y, x)$	–

TABLE I
A RUN OF THE ENUMERATIVE ALGORITHM

D. Enumerative Learning

The enumerative learning algorithm [8] adopts a dynamic programming based search strategy that systematically enumerates concepts (expressions) in increasing order of complexity. Various complexity metrics can be assigned to concepts, the simplest being the expression size. The algorithm needs to store all enumerated expressions, because expressions of a given size are composed to form larger expressions in the spirit of dynamic programming. The algorithm maintains a set of concrete test cases, obtained from the counterexamples returned by the verification oracle. These concrete test cases are used to reduce the number of expressions stored at each step by the dynamic programming algorithm.

We demonstrate the working of the algorithm on the illustrative example. Table I shows the expressions submitted to the verification oracle (an SMT solver) during the execution of the algorithm and the values for which the expression produces incorrect results. Initially, the algorithm submits the expression x to the verifier. The verifier returns a counterexample $\langle x = 0, y = 1\rangle$, corresponding to the case where the expression x violates the specification. The expression enumeration is started from scratch every time a counterexample is added. All enumerated expressions are checked for conformance with the accumulated (counter)examples before making a potentially-expensive query to the verifier. In addition, suppose the algorithm enumerates two expressions e and e' which evaluate to the same value on the examples obtained so far, then only one of e or e' needs to be considered for the purpose of constructing larger expressions.

Proceeding with the illustrative example, the algorithm then submits the expression y and the constant 1 to the verifier. The verifier returns the values $\langle x = 1, y = 0\rangle$ and $\langle x = 0, y = 0\rangle$, respectively, as counterexamples to these expressions. The algorithm then submits the expression $x+y$ to the verifier. The verifier returns the values $\langle x = 1, y = 1\rangle$ as a counterexample. The algorithm then submits the expression shown in the last row of Table I to the verifier. The verifier certifies it to be correct and the algorithm terminates.

The optimization of pruning based on concrete counterexamples helps in two ways. First, it reduces the number of invocations of the verification oracle. In the example we have described, the correct expression was examined after only four calls to the SMT solver, although about 200 expressions were enumerated by the algorithm. Second, it reduces the search space for candidate expressions significantly (see [8] for details). For instance, in the run of the algorithm on the example, although the algorithm enumerated about 200 expressions, only about 80 expressions were stored.

Production	Component	
$E \rightarrow \mathit{ITE}(B, E, E)$	Inputs:	$(i_1 : B)(i_2, i_3 : E)$
	Output:	$(o : E)$
	Spec:	$o = \mathit{ITE}(i_1, i_2, i_3)$
$B \rightarrow E \leq E$	Inputs:	$(i_1, i_2 : E)$
	Output:	$(o : B)$
	Spec:	$o = i_1 \leq i_2$

TABLE II
COMPONENTS FROM PRODUCTIONS

E. Constraint-based Learning

The symbolic CEGIS approach uses a constraint solver both for searching for a candidate expression that works for a set of concrete input examples (concept learning) and verification of validity of an expression for all possible inputs. We use component based synthesis of loop-free programs as described by Jha et al. [5], [7]. Each production in the grammar corresponds to a component in a library. A loop-free program comprising these components corresponds to an expression from the grammar. Some sample components for the illustrative example are shown in Table II along with their corresponding productions.

The input/output ports of these components are typed and only well-typed programs correspond to well-formed expressions from the grammar. To ensure this, Jha et al.'s encoding [5] is extended with typing constraints. We illustrate the working of this algorithm on the maximum of two integers example. The library of allowed components is instantiated to contain one instance each of *ITE* and all comparison operators($\leq, \geq, =$) and the concrete example set is initialized with $\langle x = 0, y = 0\rangle$. The first candidate loop-free program synthesized corresponds to the expression x. This candidate is submitted to the verification oracle which returns with $\langle x = -1, y = 0\rangle$ as a counterexample. This counterexample is added to the concrete example set and the learning algorithm is queried again. The SMT formula for learning a candidate expression is solved in an incremental fashion; i.e., the constraint for every new example is added to the list of constraints from the previous examples. The steps of the algorithm on the illustrative example are shown in Table III.

If synthesis fails for a component library, we add one instance of every operator to the library and restart the algorithm with the new library. We also tried a modification to the original algorithm [5], in which, instead of searching for a loop-free program that utilizes all components from the given library at once, we search for programs of increasing length such that every line can still select any component from the library. The program length is increased in an exponential

Iteration	Loop-free program	Learned counter-example
1	$o_1 := x$	$\langle x = -1, y = 0\rangle$
2	$o_1 := x \ \leq \ x$ $o_2 := \mathit{ITE}(o_1,\ y,\ x)$	$\langle x = 0, y = -1\rangle$
3	$o_1 := y \ \geq \ x$ $o_2 := \mathit{ITE}(o_1,\ y,\ x)$	–

TABLE III
A RUN OF THE CONSTRAINT LEARNING ALGORITHM

fashion (1, 2, 4, 8, $\cdots$) for a good coverage. This approach provides better running times for most benchmarks in our set, but it can also be more expensive in certain cases.

F. Learning by Stochastic Search

The stochastic learning procedure is an adaptation of the algorithm recently used by Schufza et al. [11] for program super-optimization. The learning algorithm of the CEGIS loop uses the Metropolis-Hastings procedure to sample expressions. The probability of choosing an expression e is proportional to a value Score(e), which indicates the extent to which e meets the specification φ. The Metropolis-Hastings algorithm guarantees that, in the limit, expressions e are sampled with probability proportional to Score(e). To complete the description of the search procedure, we need to define Score(e) and the Markov chain used for successor sampling. We define Score(e) to be $\exp(-\beta C(e))$, where β is a smoothing constant (set by default to 0.5), and the cost function $C(e)$ is the number of concrete examples on which e does *not* satisfy φ.

We now describe the Markov chain underlying the search. Fix an expression size n, and consider all expressions in L with parse trees of size n. The initial candidate is chosen uniformly at random from this set [26]. Given a candidate e, we pick a node v in its parse tree uniformly at random. Let e_v be the subexpression rooted at this node. This subtree is replaced by another subtree (of the same type) of size equal to $|e_v|$ chosen uniformly at random. Given the original candidate e, and a mutation e' thus obtained, the probability of making e' the new candidate is given by the Metropolis-Hastings acceptance ratio $\alpha(e, e') = \min(1, \text{Score}(e')/\text{Score}(e))$.

The final step is to describe how the algorithm selects the expression size n. Although the solver comes with an option to specify n, the expression size is typically not known a priori given a specification φ. Intuitively, we run concurrent searches for a range of values for n. Starting with $n = 1$, with some probability p_m (set by default to 0.01), we switch at each step to one of the searches at size $n \pm 1$. If an answer e exists, then the search at size $n = |e|$ is guaranteed to converge.

Consider the earlier example for computing the maximum of two integers. There are 768 integer-valued expressions in the grammar of size six. Thus, the probability of choosing $e = \mathit{ITE}(x \leq 0, y, x)$ as the initial candidate is $1/768$. The subexpression to mutate is chosen uniformly at random, and so the probability of deciding to mutate the boolean condition $x \leq 0$ is $1/6$. Of the 48 boolean conditions in the grammar, $y \leq 0$ may be chosen with probability $1/48$. Thus, the mutation $e' = \mathit{ITE}(0 \leq y, y, x)$ is considered with probability $1/288$. Given a set of concrete examples $\{(-1,-4),(-1,-3),(-1,-2),(1,1),(1,2)\}$, Score$(e) = \exp(-2\beta)$, and Score$(e') = \exp(-3\beta)$, and so e' becomes the new candidate with probability $\exp(-\beta)$. If, on the other hand, $e' = \mathit{ITE}(x \leq y, y, x)$ had been the mutation considered, then Score$(e') = 1$, and e' would have become the new candidate with probability 1.

Our algorithm differs from that of Schufza et al. [11] in three ways: (1) we do not attempt to optimize the size of the expression while the super-optimizer does so; (2) we synthesize expression graphs rather than straight-line assembly code, and (3) since we do not know the expression size n, we run concurrent searches for different values of n, whereas the super-optimizer can use the size of the input program as an upper bound on program size.

IV. Benchmarks and Evaluation

We are in the process of assembling a benchmark suite of synthesis problems to provide a basis for side-by-side comparisons of different solution strategies. The current set of benchmarks is limited to synthesis of loop-free functions with no optimality criterion; nevertheless, the benchmarks provide an initial demonstration of the expressiveness of the base formalism and of the relative merits of the individual solution strategies presented earlier. Specifically, in this section we explore three key questions about the benchmarks and the prototype synthesizers.

- **Complexity of the benchmarks.** Our suite includes a range of benchmarks from simple toy problems to non-trivial functions that are difficult to derive by hand. Some of the benchmarks can be solved in a few hundredths of a second, whereas others could not be solved by any of our prototype implementations. In all cases, however, the complexity of the problems derives from the size of the space of possible functions and not from the complexity of checking whether a candidate solution is correct.
- **Relative merits of different solvers.** The use of a standard format allows us to perform the first side-to-side comparison of different approaches to synthesis. None of the implementations were engineered with high-performance in mind, so the exact solution times are not necessarily representative of the best that can be achieved by a particular approach. However, the order of magnitude of the solution times and the relative complexity of the different approaches on different benchmarks can give us an idea of the relative merits of each of the approaches described earlier.
- **Effect of problem encoding.** For many problems, there are different natural ways to encode the space of desired functions into a grammar, so we are interested in observing the effect of these differences in encoding for the different solvers.

To account for variability and for the constant factors introduced by the prototype nature of the implementations, we report only the order of magnitude of the solution times in five different buckets: 0.1 for solution times less than half a second, 1 for solution times between half a second and 15 seconds, 100 for solution times up to two minutes, 300 for solution times of up to 5 minutes, and infinity for runs that time out after 5 minutes.

The benchmarks themselves are grouped into three categories: hacker's delight problems, integer benchmarks, and assorted boolean and bit-vector problems.

Hacker's delight benchmarks: This set includes 57 different benchmarks derived from 20 different bit-manipulation problems from the book *Hacker's Delight* [27]. These bit-vector problems were among the first to be successfully tackled by synthesis technology and remain an active area of

Fig. 2. Selected performance results for the three classes of benchmarks

research [4], [5], [16]. For these benchmarks, the goal is to discover clever implementations of bit-vector transformations (colloquially known as bit-twiddling). For most problems, there are three different levels of grammars numbered $d0$, $d1$ and $d5$; level $d0$ involves only the instructions necessary for the implementation, so the synthesizer only needs to discover how to connect them together. Level $d5$, on the other extreme, involves a highly unconstrained grammar, so the synthesizer must discover which operators to use in addition to how to connect them together.

Fig. 2 shows the performance of the three solvers on a sample of the benchmarks. For the Hacker's Delight benchmarks (`hd`) we see that the enumerative solver dominates, followed by the stochastic solver. The symbolic search was the slowest, failing to terminate on 29 of the 57 benchmarks. It is worth mentioning, however, that none of the grammars for these problems required the synthesizer to discover the bit-vector constants involved in the efficient implementations. We have some evidence to suggest that the symbolic solver can discover such constants from the full space of 2^{32} possible constants with relatively little additional effort. On the other hand, for many of these problems the magic constants come from a handful of values such as 1, 0, or `0xffffffff`, so it is unnecessary for the enumerative solver to search the space of 2^{32} possible bit-vectors.

Finally, because these benchmarks have different grammars for the same problem, we can observe the effect of using more restrictive or less restrictive grammars as part of the problem description. We can see in the data that all solvers were affected by the encoding of the problem for at least some benchmark; although in some cases, the pruning strategies used by the solvers were able to ameliorate the impact of the larger search space.

Integer benchmarks: These benchmarks are meant to be loosely representative of synthesis problems involving functions with complex branching structures involving linear integer arithmetic. One of the benchmarks is `array-search`, which synthesizes a loop-free function that finds the index of an element in a sorted tuple of size n, for n ranging from 2 to 16. This benchmark proved to be quite complex, as no solver was able to synthesize this function for $n > 4$. The `max` benchmarks are similar except they compute the maximum of a tuple of size n.

Fig. 2 shows the relative performance of the three solvers on these benchmarks for sizes up to 4. With one exception, the enumerative solver is the fastest for this class of benchmarks, followed by the stochastic solver. The exception was `max3` where the stochastic solver was faster.

Boolean/Bit-vector benchmarks: The `parity` benchmark computes the parity of a set of Boolean values. The different versions represent different grammars to describe the set of Boolean functions. As with other benchmarks, the enumerative solver was always faster, whereas the symbolic solver failed on every instance. These results show the impact that different encodings of the same space of functions can have on the solution time for both of the solution strategies that succeeded. Unlike the `hd` benchmarks where the different grammars for a given benchmark were strict subsets of each other, in this case the encodings `AIG` and `NAND` correspond to different representations of the same space of functions.

The `Morton` benchmarks, which involve the synthesis of a function to compute Morton numbers, are intended as challenge problems, and could not be completed by any of the synthesizers.

Observations: The number of benchmarks and the maturity of the solvers are too limited to draw broad conclusions,

but the overall trend we observe is that the encoding of the problem space into grammar has a significant impact on performance, although the solvers are often good at mitigating the effect of larger search spaces. We can also see that non-symbolic techniques can be effective in exploring spaces of implementations and can surpass symbolic techniques, especially when the problems do not require the synthesizer to derive complex bit-vector constants, which is true for all the bit-vector benchmarks used. Moreover, we observe that the enumerative technique was better than the stochastic search for all but two benchmarks, so although both implementations are immature, these results suggest that it may be easier to derive good pruning rules for the explicit search than an effective fitness function for the stochastic solver.

The symbolic solver used for these experiments represents one of many possible approaches to encoding the synthesis problem into a series of constraints. We have some evidence that more optimized encodings can make the symbolic approach more competitive, although there are still many problems for which the enumerative approach is more effective. Specifically, we have transcribed all the hacker's delight and integer benchmarks into the input language of the Sketch synthesis system [24]. Sketch completed all but 11 of the `hd` benchmarks, and it was able to synthesize `array-search` up to size 7. This experiment is not an entirely fair comparison because, although Sketch uses a specialized constraint solver and carefully tuned encodings, the symbolic solver presented in this paper uses a direct encoding of the problem into sequences of constraints and uses Z3, a widely used off-the-shelf SMT solver which is not as aggressively tuned for synthesis problems. Despite these limitations, the symbolic solver was able to solve many of the benchmarks, providing a lower bound on what can be achieved with a straightforward use of off-the-shelf technology.

Moreover, the enumerative solver was able to solve more `hd` problems than even the more optimized symbolic solver. The problems where the enumerative solver succeeded but Sketch failed were the $d5$ versions of problems 11, 12, 14 and 15, which suggests that the enumerative solver was better at pruning unnecessary instructions from the grammar. On the other hand, the more optimized symbolic solver did have a significant advantage in the `array-search` benchmarks which the enumerative solver could only solve up to size 4.

V. Conclusions

Aimed at formulating the core computational problem common to many recent tools for program synthesis in a canonical and logical manner, we have formalized the problem of syntax-guided synthesis. Our prototype implementation of the three approaches to solve this problem is the first attempt to compare and contrast existing algorithms on a common set of benchmarks. We are already working on the next steps in this project. These consist of (1) finalizing the input syntax (SYNTH-LIB) based on the input format of SMT-LIB2, with an accompanying publicly available parser, (2) building a more extensive and diverse repository of benchmarks, and (3) organizing a competition for SyGuS-solvers. We welcome feedback and help from the community on all of these steps.

Acknowledgements: We thank Nikolaj Bjorner and Stavros Tripakis for their feedback. This research is supported by the NSF Expeditions in Computing project ExCAPE (award CCF 1138996).

References

[1] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009, vol. 4, ch. 8.
[2] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, 2011.
[3] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM TOPLAS*, vol. 2, no. 1, pp. 90–121, 1980.
[4] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *PLDI*, 2005.
[5] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," *SIGPLAN Not.*, vol. 46, pp. 62–73, June 2011.
[6] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.
[7] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010, pp. 215–224.
[8] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: Specifying Protocols with Concolic Snippets," in *PLDI*, 2013, pp. 287–296.
[9] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL*, 2010, pp. 313–326.
[10] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Software synthesis procedures," *Commun. ACM*, vol. 55, no. 2, pp. 103–111, 2012.
[11] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ASPLOS*, 2013, pp. 305–316.
[12] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear invariant generation using non-linear constraint solving," in *CAV*, 2003, pp. 420–432.
[13] A. Rybalchenko, "Constraint solving for program verification: Theory and practice by example," in *CAV*, 2010, pp. 57–71.
[14] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *ESOP*, 2013, pp. 574–592.
[15] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "On solving universally quantified Horn clauses," in *SAS*, 2013, pp. 105–125.
[16] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.
[17] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.
[18] P. D. Summers, "A methodology for LISP program construction from examples," *J. ACM*, vol. 24, no. 1, pp. 161–175, 1977.
[19] E. Y. Shapiro, *Algorithmic Program Debugging*. Cambridge, MA, USA: MIT Press, 1983.
[20] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, pp. 237–269, Sep. 1983.
[21] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.
[22] S. A. Seshia, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *DAC*, 2012, pp. 356–365.
[23] D. Angluin, "Queries and concept learning," *Machine Learning*, vol. 2, pp. 319–342, 1988.
[24] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California, Berkeley, 2008.
[25] 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.
[26] B. McKenzie, "Generating strings at random from a context free grammar," 1997.
[27] H. S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Tutorial: Practical Verification of Network Programs

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts, Amherst

Mark Reitblatt
Cornell University

Cole Schlesinger
Princeton University

I. Traditional Networking

Computer networks are essential infrastructure in modern society. Much like the electric power grid, we expect networks to always function, and there are often serious material consequences when they fail. Unfortunately, network failures are all too common. At Amazon, a configuration error during routine maintenance triggered cascading failures that shut down a datacenter and the customer machines hosted there. At GoDaddy, a corrupted routing table disabled their domain name service (DNS) for a day, causing a widespread outage. At United Airlines, a network connectivity issue disabled their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub. Even worse, each of these failures could have been avoided—they were all caused by *operator errors* or *software bugs* [6], [13], [22].

The high rate of network failures should not be surprising. A typical datacenter or enterprise network is a complex system with thousands of devices: routers and switches, web caches and load balancers, monitoring middleboxes and firewalls, and more. Each type of device runs a stack of interrelated protocols and is configured by idiosyncratic, vendor-specific interfaces. Network operators have to grapple with this complexity to implement high-level, end-to-end policies. For example, an access control policy or a quality of service guarantee may need to be implemented by stringing together configurations on several devices. Network operators who can accomplish these feats have been called "masters of complexity" [18], for good reason!

The complexity of traditional networks has also made it extremely difficult to build automated tools for reasoning precisely about end-to-end behavior. To make an effective tool, one would need to somehow reverse-engineer the semantics of numerous poorly-documented devices, construct parsers for proprietary protocols, and formalize their concurrent execution and asynchronous interactions. Although formal models of traditional networks have been developed, they are either too complex to be effective or too abstract to be practical.

II. Software-Defined Networking

Recently, a new network architecture has emerged called *software defined networking* (SDN) that addresses the many of the issues listed above. An SDN eliminates the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, *etc.*—and replaces them with commodity *programmable switches*. These switches are managed and programmed by a logically-centralized *controller* machine, which communicates with switches using a standard protocol such as *OpenFlow* [14].

Since OpenFlow-programmable switches conform to a well-defined interface, it is possible to reason about their behavior and even build formal models of their operation. This has sparked a lot of interest in building verification tools for software defined networks. Before introducing verification, this tutorial will start with begin with an introduc to OpenFlow itself. Using Ox, a simple, OCaml-based controller, participants will first learn how to write some simple SDN applications. The skills they learn will be directly applicable to other popular platforms, such as NOX [7], POX [17], Beacon [3], Nettle [19], and Floodlight [5].

III. Programming with Frenetic

OpenFlow and SDN make network programming *possible*, but they do not make it easy. The first part of the tutorial will make it evident that the OpenFlow abstraction is quite low-level; although it abstracts away several hardware details, it still feels like an "assembly language" for switch programming. It is particularly hard to run several programs or modules simultaneously when programming directly with OpenFlow. If composed naively, two applications are almost certain to destroy each others' network state. Broadly, OpenFlow itself lacks the mechanisms that we need to construct software from separate, modular components.

To address this issue, we will introduce *Frenetic*, a high-level language for programming SDN. Unlike OpenFlow, which requires programmers to carefully manipulate low-level switch-state, Frenetic provides a much higher level of abstraction: a Frenetic program denotes a mathematical, packet-processing function. Frenetic provides a collection of simple functions for filtering, modifying, counting, and forwarding packets, as well as several operators that combine smaller functions into larger ones. The Frenetic compiler takes care of translating these functions into low-level OpenFlow instructions, and the Frenetic runtime system addresses several other details of OpenFlow.

In this tutorial, we will show participants how to program SDNs in a modular way, using Frenetic's abstractions. We will build several realistic network applications from the ground up, and also learn to use more sophisticated modules, such as NAT and MAC-learning, which are part of the Frenetic standard library. We will also look under the hood to see how the Frenetic compiler and runtime system work.

Although the tutorial will focus on Frenetic, we hope to impart an understanding of other *network programming*

ISBN 978-0-9835678-3-7/13.

languages, such as Pyretic [15], Maple [20], and PANE [4]. Although these languages provide a variety of abstractions, they all address issues of modularity and composition that Frenetic also tackles.

IV. Verification with Frenetic

Frenetic's modularity and composition operators make SDN programming much easier; however, SDN promises to make networks verifiable, too. There are several verification tools that operate directly on low-level network state [1], [10], [12], [11], but Frenetic programs can be verified at the source-level.

This tutorial will introduce the Frenetic verification tool, which can check *reachability properties* of source-level Frenetic programs automatically. This tool enables programmers to automatically answer questions such as, "is host A reachable from host B?", "is there a loop involving C?", "is all SSH traffic blocked?", and so on. These are precisely the kinds of questions that network operators ask whilst debugging and troubleshooting their networks.

Under the hood, the Frenetic verification tool operates by encoding programs and properties as SAT formulae and checks their satisfiability using the Z3 theorem prover. Thanks to Frenetic's well-defined, high-level semantics, the encoding is fairly straightforward and certainly much simpler than tools that work with OpenFlow directly.

V. Conclusion

We hope this tutorial will show you how programming languages technology and formal methods can be used to both build networks and verify important network properties. Since this is an in-depth, hands-on tutorial, we will only get to use a small selection of tools and technologies, developed as part of the Frenetic project. However, your experience with Frenetic and its tools will also help you understand the many other languages and tools that have been developed for this domain.

Acknowledgments: Our work is supported in part by the National Science Foundation under grant CNS-1111698, the Office of Naval Research under award N00014-12-1-0757, a Sloan Research Fellowship, and a Google Research Award.

References

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), Aug 2009.

[3] David Erickson. The Beacon OpenFlow controller. In *HotSDN*, 2013.

[4] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[5] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[6] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[7] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[8] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM*.

[9] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2010.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[12] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[13] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational IP backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, Aug 2008.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[15] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, Apr 2013.

[16] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[17] The POX OpenFlow controller, Jul 2011. Available from http://www.noxrepo.org/pox/about-pox.

[18] Scott Shenker, Martin Casado, Teemu Koponen, and Nick McKeown. The future of networking and the past of protocols, Oct 2011. Invited talk at Open Networking Summit.

[19] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[20] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[21] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE), Boston, MA*, Mar 2011.

[22] Zuoning Yin, Matthew Caesar, and Yuanyuan Zhou. Towards understanding bugs in open source router software. In *SIGCOMM CCR*, 2010.

Firmware Validation: Challenges and Opportunities

Jim Grundy
Intel Corporation

ABSTRACT

Firmware validation is driven by imperatives and challenges distinct from those of application level software. In this tutorial we will survey the characteristics of firmware projects, focusing on those that make them particularly challenging and important to validate. Well look at the tasks accomplished using firmware, the environments in which it executes, and how firmware is shaped by the constraints imposed by the greater product development program in which it fits. Finally, well look at some of our experiences in firmware validation and the lessons weve learned from them. Specifically, well be looking for lessons that can help to guide the selection of problems to study and appropriate case studies on which to evaluate them.

SHORT BIOGRAPHY

Jim Grundy is a research scientist with the Strategic CAD Labs at Intel Corporation, where he leads the Logic Verification group in developing formal tools and methods for modeling and analysis of designs to be realized in both hardware and software. He has published in the fields of automated and interactive reasoning, software verification, and functional programming. Prior to joining Intel in 2000, Jim was faculty a member of the Department of Computer Science at The Australian National University. Jim has also worked as a post-doctoral researcher at bo Akademi in Finland, and as a research scientist at the Australian Defence Science and Technology Organisation. Jim holds a PhD from the University of Cambirdge, UK and BSc from the University of Queensland in Australia.

ISBN 978-0-9835678-3-7/13.

Secure Programs via Game-based Synthesis

Somesh Jha
Tom Reps
Bill Harris
University of Wisconsin (Madison)

ABSTRACT

Several recent operating systems provide system calls that allow an application to explicitly manage the privileges of modules with which the application interacts. Such privilege-aware operating systems allow a programmer to a write a program that satisfies a strong security policy, even when the program interacts with untrusted modules. However, it is often non-trivial to rewrite a program to correctly use the system calls to satisfy a high-level security policy.

This paper concerns the policy-weaving problem, which is to take as input a program, a desired high-level policy for the program, and a description of how system calls affect privilege, and automatically rewrite the program to invoke the system calls so that it satisfies the policy. We describe a reduction from the policy-weaving problem to finding a winning strategy to a two-player safety game. We then describe a policy-weaver generator that implements the reduction and a novel game-solving algorithm, and present an experimental evaluation of the generator applied to a model of the Capsicum capability system. We conclude by outlining ongoing work in applying the generator to a model of the HiStar decentralized-information-flow control (DIFC) system.

SHORT BIOGRAPHIES

Somesh Jha received his B.Tech from Indian Institute of Technology, New Delhi in Electrical Engineering. He received his Ph.D. in Computer Science from Carnegie Mellon University in 1996. Currently, Somesh Jha is a Professor in the Computer Sciences Department at the University of Wisconsin (Madison), which he joined in 2000. His work focuses on analysis of security protocols, survivability analysis, intrusion detection, formal methods for security, and analyzing malicious code. Recently he has also worked on privacy-preserving protocols. Somesh Jha has published over 100 articles in highly-refereed conferences and prominent journals. He has won numerous best-paper awards. Somesh also received the NSF career award in 2005.

Thomas W. Reps is Professor of Computer Science in the Computer Sciences Department of the University of Wisconsin, which he joined in 1985. Reps is the author or co-author of four books and more than one hundred seventy-five papers describing his research (see http://www.cs.wisc.edu/ reps/). His work has concerned a wide variety of topics, including program slicing, dataflow analysis, pointer analysis, model checking, computer security, code instrumentation, language-based program-development environments, the use of program profiling in software testing, software renovation, incremental algorithms, and attribute grammars.

His collaboration with Professor Tim Teitelbaum at Cornell University from 1978 to 1985 led to the creation of two systems the Cornell Program Synthesizer and the Synthesizer Generator that explored how to build interactive programming tools that incorporate knowledge about the programming language being supported. The systems that they created were similar to modern program-development environments, such as Microsoft Visual Studio and Eclipse, but pre-dated them by more than two decades. Reps is President of GrammaTech, Inc., which he and Teitelbaum founded in 1988 to commercialize this work.

Since 1985, Professor Reps has been co-leader, with Professor Susan Horwitz, of a research group at the University of Wisconsin that has carried out many investigations of program slicing and its applications in software engineering. His most recent work concerns program analysis, computer security, and software model checking.

In 1996, Reps served as a consultant to DARPA to help them plan a project aimed at reducing the impact of the Year 2000 Problem on the U.S. Department of Defense. In 2003, he served on the F/A-22 Avionics Advisory Team, which provided advice to the U.S. Department of Defense about problems uncovered during integration testing of the planes avionics software.

Professor Reps received his Ph.D. in Computer Science from Cornell University in 1982. His Ph.D. dissertation won the 1983 ACM Doctoral Dissertation Award.

Repss 1988 paper on interprocedural slicing, with Susan Horwitz and his then-student David Binkley, was selected as one of the 50 most influential papers from the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1979-99. According to Google Scholar, the 1988 paper and the subsequent journal version have received over 1,600 citations.

His 2004 paper about analysis of assembly code, with his student Gogul Balakrishnan, received the ETAPS Best-Paper Award for 2004 from the European Association for Programming Languages and Systems (EAPLS). His 2008 paper about a system for generating static analyzers for machine instructions, with his student Junghee Lim, received the ETAPS Best-Paper

ISBN 978-0-9835678-3-7/13.

Award for 2008 from EAPLS. In 2010, his 1984 paper ”The Synthesizer Generator”, with Tim Teitelbaum, received an ACM SIGSOFT Retrospective Impact Paper Award. In 2011, his 1994 paper ”Speeding up slicing”, with Susan Horwitz, Mooly Sagiv, and Genevieve Rosay, also received an ACM SIGSOFT Retrospective Impact Paper Award.

Three of his students, Gogul Balakrishnan, Akash Lal, and Junghee Lim have been the recipients of the Outstanding Graduate Student Research Award given by the University of Wisconsin Computer Sciences Department. Akash Lal was also a co-recipient of the 2009 SIGPLAN Outstanding Doctoral Dissertation Award, and he was named as one of the 18 awardees selected for the 2011 India TR-35 list (top innovators under 35).

In 2003, Reps was recognized as a ”Highly Cited Researcher” in the field of Computer Scienceone of 230 worldwide who received such recognition by the Institute for Scientific Information. As of February 2013, Reps was ranked 8th (citations) and 4th (field rating) on Microsoft Academic Searchs list of most-highly-cited authors in the field of Programming Languages, and 23rd (citations) and 13th (field rating) on its list of most-highly-cited authors in the field of Software Engineering.

Reps has also been the recipient of an NSF Presidential Young Investigator Award (1986), a Packard Fellowship (1988), a Humboldt Research Award (2000), and a Guggenheim Fellowship (2000). He is also an ACM Fellow (2005).

Reps has held visiting positions at the Institut National de Recherche en Informatique et en Automatique (INRIA) in Rocquencourt, France (1982-83), the University of Copenhagen, Denmark (1993-94), the Consiglio Nazionale delle Ricerche in Pisa, Italy (2000-2001), and the University Paris Diderot Paris 7 (2007-2008).

William Harris is a PhD student and research assistant at the University of Wisconsin-Madison, where he is advised by Somesh Jha and Thomas Reps. His current research focuses on applying formal methods to problems in computer security. He received his B.S. from Purdue University in 2007, and received his M.S. from the University of Wisconsin-Madison in 2011. He has worked as a visiting researcher for NEC Labs America and Microsoft Research. He was a Microsoft Research Fellow from 2010 - 2011.

Using Process Modeling and Analysis Techniques to Reduce Errors in Healthcare

Lori A. Clarke
University of Massachusetts, Amherst

ABSTRACT

As has been widely reported in the news lately, healthcare errors are a major cause of death and suffering. In the University of Massachusetts Medical Safety Project, we are exploring the use of process modeling and analysis technologies to help reduce medical errors and improve efficiency. Specifically, we are modeling healthcare processes using a process definition language and then analyzing these processes using model checking, fault-tree analysis, discrete event simulation, and other techniques. Working with the UMASS School of Nursing and the Baystate Medical Center, we are undertaking in-depth case studies on error-prone and life-critical healthcare processes. In many ways, these processes are similar to complex, distributed systems with many interacting, concurrent threads and numerous exceptional conditions that must be handled carefully.

This talk describes the technologies we are using, discusses case studies, and presents our observations and findings to date. Although presented in terms of the healthcare domain, the described approach could be applied to human-intensive processes in other domains to provide a technology-driven approach to process improvement.

SHORT BIOGRAPHY

Lori A. Clarke is chair of the School of Computer Science at the University of Massachusetts, Amherst and is co-director of the Laboratory for Advanced Software Engineering Research (LASER). She is a Fellow of the ACM and IEEE, and a board member of the Computing Research Associations Committee on the Status of Women in Computing Research (CRA-W). She received the 2012 ACM Special Interest Group on Software Engineering (SIGSOFT) Outstanding Research Award, the 2011 University of Massachusetts Outstanding Accomplishments in Research and Creative Activity Award, the 2009 College of Natural Sciences and Mathematics Outstanding Faculty Service Award, the 2004 University of Colorado, Boulder Distinguished Engineering Alumni Award, the 2002 SIGSOFT Distinguished Service Award, a 1993 University Faculty Fellowship, and the 1991 University of Massachusetts Distinguished Faculty Chancellors Medal. She is a former vice chair of the Computing Research Association (CRA), co-chair of CRA-W, IEEE Publication Board member, associate editor of ACM Transactions on Programming Languages and Systems (TOPLAS) and IEEE Transactions on Software Engineering (TSE), member of the CCR NSF advisory board, ACM SIGSOFT secretary/treasurer, vice-chair and chair, IEEE Distinguished Visitor, and ACM National Lecturer. She has written numerous papers, served on many program committees, and was program co-chair of the 14th and general chair of the 25th International Conference on Software Engineering. She has been a Principal Investigator on a number of NSF, DoD, and DARPA projects.

Dr. Clarkes research is in the area of software engineering, primarily focusing on verification and requirements engineering for human-intensive systems. She has been investigating techniques for detecting errors and safety and security vulnerabilities in complex processes in domains such as healthcare and digital government. She is also involved in several efforts to increase participation of underrepresented groups in computing research.

ISBN 978-0-9835678-3-7/13.

Static Verification Based Signoff – A Key Enabler for Managing Verification Complexity in the Modern SoC

Pranav Ashar
Real Intent

ABSTRACT

Application-based verification, *i.e.*, partitioning the verification process by verification concerns, has become an important approach for managing verification complexity in the billion-transistor SoC. This new verification paradigm has truly come into focus with the proliferation of layers of complexity in an SoC beyond the baseline complexity of its constituent components. In a sense, the nature of chip complexity has shifted from how much goes into a chip to what goes into a chip. Given a narrow verification concern like clock-domain verification, power, dft, reset analysis etc, the specification, analysis and debug dimensions of the verification problem become meaningfully solvable. This is a new paradigm in a sense because it focuses technologists toward the development of complete solutions and closure for the problem at hand as a whole rather than on just nuts-and-bolts technologies like simulation and ABV. Static formal analysis is able to play a key role in this paradigm for various reasons. With the narrow focus on a specific verification problem, much of the specification becomes precise and implicit. In addition, the limited scope allows the formal analysis to be controlled and nominally tractable. Further, even when the formal analysis remains bounded, it is still possible to return actionable information to the user. Finally, debug becomes much more precise and actionable in the context of the narrow verification concern being addressed. These aspects all come to fore in the verification of clock domain crossings in the modern SoC. Used to be that a chip would have a handful of clock domains and the clock-domain checking could be done manually. With 100s of clocks domains on chip, that luxury is not available any more. No SoC gets taped out today without a dedicated sign-off of clock-domain crossings using verification tools specialized for this problem. Another reason clock-domain verification is good to highlight as an example of the new paradigm is that it is at the intersection of chip functionality and timing. This verification task cannot be completed by just functional simulation or just by static timing analysis. It needs a specialized solution, with static formal analysis at its core, to do justice to it.

SHORT BIOGRAPHY

Dr. Pranav Ashar is the Chief technology Officer at Real Intent Inc., a System-on-Chip verification company. Pranav received his Ph.D. in EECS with emphasis on EDA from the University of California, Berkeley in 1991. He was then at NEC Labs in Princeton, NJ for 13 years where he developed a number of EDA technologies that have influenced the industry. Pranav has authored about 70 refreed publications with more than 1000 citations, and co-authored a book titled ”Sequential Logic Synthesis”. He has 35 patents granted and pending. Pranav was an adjunct faculty in the CSEE department at Columbia University where he has taught graduate and undergraduate courses on VLSI design automation, VLSI Verification, and VLSI design.

ISBN 978-0-9835678-3-7/13.

The FMCAD Graduate Student Forum

Thomas Wahl, Student Forum Chair
Northeastern University, College of Computer and Information Science
Boston, Massachusetts 02115

FMCAD 2013 featured an event new to the FMCAD conference series, the Graduate Student Forum, held on Monday October 21, following the joint MEMOCODE/FMCAD Tutorial Day. The intention of the Forum was to specifically attract students to the conference, by providing them with a platform for introducing their research to the wider Formal Methods community, and obtain feedback on it. Submissions were solicited in the form of short reports describing research ideas, or ongoing work in the scope of the FMCAD conference that the student is currently pursuing.

In response to the Call for Student Forum papers, the Organizing Committee received 29 submissions (after discounting a few withdrawals), many of them of very high quality. As expected for a first-time (rather than a "broken-in") event, the flavor of submissions varied considerably, from initial reports on brand-new research, to extensions of work published before, and survey-style articles summarizing previous results by the author. The submissions also varied greatly by topic, ranging from software model checking and HW/SW co-verification, to foundational papers on automata theory and games, to micro-architecture verification and behavioral hardware synthesis.

The submissions were reviewed by members of a small subset of the FMCAD 2013 Program Committee. After an initial review by one committee member, each submission was discussed in detail by the Student Forum sub-committee. The reviews focused on novelty of the work, the technical maturity of the submission, and on the quality of the presentation. While both proposed future work as well as already conducted work was acceptable, in either case some (unpublished) novel insight or contribution was expected. Following the discussion, a total of 14 submissions were accepted for inclusion in the FMCAD 2013 program, amounting to an acceptance rate of around 50%. The list of accepted submissions can be found on the following page in these proceedings.

The Student Forum itself consisted of very short presentations by the student authors of each accepted submission, and of a poster that was on display throughout the duration of FMCAD.

The FMCAD Student Forum Chair wishes to express his sincere gratitude to FMCAD's sponsors for their very generous support of the event, in particular via a substantial financial contribution from FMCAD Inc., via several travel grants made available by NVIDIA Corporation directly for the Student Forum, and via further financial support by Atrenta Inc., Galois Inc., IBM Corporation, Intel Corporation, Jasper Design Automation, Mentor Graphics, Microsoft Corporation, NEC Labs America, OneSpin Solutions, Oski Technology Inc., Real Intent, Synopsys, and Xpliant. Thanks to this support, it was possible to provide a large number of student participants with a substantial travel grant. The FMCAD Graduate Student Forum would not have been possible in this format without this extensive sponsorship.

The Student Forum would also not have been possible without the hard work by the student authors and their many excellent submissions. The Chair is grateful to all contributors to this successful event, especially to the FMCAD 2013 General and Program Chairs, Barbara Jobstmann and Sandip Ray, for their advice, assistance, and encouragement.

Portland, October 21, 2013

Thomas Wahl
Student Forum Chair

List of Accepted Papers

The following list shows the names of the author(s) and the title of each of the accepted FMCAD 2013 Student Forum submissions, in alphabetical order of the last name of the first author of each submission.

[1] **Kai Cong.** Symbolic Execution of Virtual Devices

[2] **Eva Darulova.** Programming with Uncertainties

[3] **Marko Dimjasevic.** Automatic Testing of Software Libraries

[4] **Shilpi Goel.** A Formal Model for Machine Code Proofs

[5] **Mitesh Jain and Panagiotis Manolios.** Skipping Refinement

[6] **Sebastiaan Joosten and Julien Schmaltz.** Automated Deadlock Verification in Register Transfer Level Designs of Communication Fabrics

[7] **Jan Lanik.** Low-Power gate decomposition for spatially correlated temporal-dependent input vectors

[8] **Li Lei.** Hardware/Software Co-monitoring

[9] **Lihao Liang.** Effective Verification of Low-Level Software with Nested Interrupts

[10] **Peizun Liu and Zhaoliang Liu.** On-the-fly Parameterized Boolean Program Exploration

[11] **Disha Puri.** Towards Certifiable Loop Pipelining Transformations in Behavioral Synthesis

[12] **Arjun Radhakrishna.** A Case for Quantitative Specifications

[13] **Bernard van Gastel and Julien Schmaltz.** Formal Verification of Communication Fabrics Micro-Architectures

[14] **Zhenkun Yang.** Compiler Transformation Validation in Behavioral Synthesis

Distributed Synthesis for LTL Fragments

Krishnendu Chatterjee, Thomas A. Henzinger, Jan Otop, Andreas Pavlogiannis
IST Austria
{chatterjee, tah, jotop, pavlogiannis}@ist.ac.at

***Abstract*—We consider the distributed synthesis problem for temporal logic specifications. Traditionally, the problem has been studied for LTL, and the previous results show that the problem is decidable iff there is no information fork in the architecture. We consider the problem for fragments of LTL and our main results are as follows: (1) We show that the problem is undecidable for architectures with information forks even for the fragment of LTL with temporal operators restricted to next and eventually. (2) For specifications restricted to globally along with non-nested next operators, we establish decidability (in EXPSPACE) for star architectures where the processes receive disjoint inputs, whereas we establish undecidability for architectures containing an information fork-meet structure. (3) Finally, we consider LTL without the next operator, and establish decidability (NEXPTIME-complete) for all architectures for a fragment that consists of a set of safety assumptions, and a set of guarantees where each guarantee is a safety, reachability, or liveness condition.**

I. Introduction

Synthesis and distributed synthesis. The *synthesis* problem is the most rigorous form of systems design, where the goal is to construct a system from a given temporal logic specification. The problem was originally proposed by Church [1] for synthesis of circuits, and has been revisited in many different contexts, such as supervisory control of discrete event systems [2], synthesis of reactive modules [3], and several others. In a seminal work, Pnueli and Rosner [4] extended the classical synthesis problem to a distributed setting. In the *distributed synthesis* problem, the input consists of (i) an *architecture* of synchronously communicating processes, that exchange messages through communication channels; and (ii) a *specification* given as a temporal logic formula; and the synthesis question asks for a reactive system for each process such that the specification is satisfied. The most common logic to express the temporal logic specification is the linear-time temporal logic (LTL) [5].

Previous results for distributed synthesis for LTL. In general the distributed synthesis problem is undecidable for LTL, but the problem is decidable for pipeline architectures [4]. The undecidability proof uses ideas originating from the undecidability proof of three-player imperfect-information games [6], [7]. The decidability results for distributed synthesis have been extended to other similar architectures, such as one-way rings [8], and also a distributed games framework was proposed in [9]. Finally, a complete topological criterion on the architecture for decidability of distributed synthesis for LTL was presented [10], where it was shown that the problem is decidable if and only if there is no *information fork* in the underlying architecture. Architectures without information forks can essentially be reduced to pipelines.

Fragments of LTL. While LTL provides a very rich framework to express temporal logic specifications, in recent years, several fragments of LTL have been considered for efficient synthesis of systems in the non-distributed setting. Such fragments often encompass a large class of properties that arise in practice and admit efficient synthesis algorithms, as compared to the whole LTL. In [11], [12] the authors considered a fragment of LTL with only *eventually* (reachability) and *globally* (safety) as the temporal operators. In [13] LTL with only eventually and globally operators (but without *next* and *until* operators) was considered for efficient translation to deterministic automata. The temporal logic specifications for reactive systems often consist of a set of assumptions and a set of guarantees, and the reactive system must satisfy the guarantees if the environment satisfies the assumptions. In [14] the GR1 (generalized reactivity 1) fragment of LTL was introduced where each assumption and guarantee is a liveness condition; and it has been shown that GR1 synthesis is very effective to automatically synthesize industrial protocols such as the AMBA protocol [15], [16].

Our contributions. In this work we consider the distributed synthesis problem for fragments of LTL. The previous results in the literature considered the whole LTL and characterized architectures that lead to decidability of distributed synthesis. In contrast, we consider fragments of LTL to present finer characterizations of the decidability results. Our main contributions are as follows:

1) *Reachability properties.* First we consider the fragment of LTL with next and eventually (reachability) as the only temporal operators, and establish that the distributed synthesis problem is undecidable if there is an information fork in the underlying architecture. In particular, the problem is undecidable with one nesting depth of the next operator and only one eventually operator; i.e., if we consider the fragment of LTL that consists of Boolean combinations of atomic propositions and next of atomic propositions; and only one eventually as the temporal operator, then the distributed synthesis problem is undecidable iff there is an information fork in the architecture.
2) *Safety properties.* We then consider the fragment of LTL with next and globally (safety) as the only temporal operators, with a single occurrence of the globally operator. We show that the distributed synthesis problem can be decidable under the existence of information forks; in particular we establish decidability (in EXPSAPCE) for the star architecture where processes have no common inputs from the environment. However, we show that

ISBN 978-0-9835678-3-7/13.

the problem remains undecidable for architectures containing an *information fork-meet*, a structure in which two processes receive sets of disjoint inputs, (as in the information fork case), and a third process receives the union of those sets. Moreover, our undecidability proof again uses specifications that do not contain nested next operators. In other words, if there is information fork, the problem may be decidable, but if there is information fork, and then the forked information meets again, then we obtain undecidability.

3) *Temporal specifications without the next operator.* Since our results show that even with one nesting depth of the next operator, distributed synthesis is undecidable with reachability and safety objectives, we finally consider the problem without the next operator. We show that if we consider a set of safety assumptions, and a set of guarantees such that each guarantee is a safety, reachability, or a liveness guarantee, then the distributed synthesis problem is decidable (and NEXPTIME-complete) for *all* architectures.

Hence, our paper improves upon existing results by presenting finer (un)decidability characterizations of the distributed synthesis problem for fragments of LTL. We also remark that when we establish decidability, it is either EXPSPACE or NEXPTIME-complete, as compared to previous proofs of decidability in distributed synthesis setting where the complexity is non-elementary. Thus as compared to the complexity of previous decidability results (tower of exponentials), our complexities (at most two exponentials) are very modest.

II. Model description

Architectures. An *architecture* is a tuple $\mathcal{A} = (\mathcal{P}, p_e, V, E)$, where $\mathcal{P} = \{p_e, p_1, p_2, \dots p_n\}$ is a set of $n+1$ *processes*, p_e is a distinguished process representing the *environment*, V is a set of (output) binary *variables*, and $E : \mathcal{P} \times \mathcal{P} \to 2^V$ defines the communication variables between processes (i.e, $E(p,q) = \{u, v\}$ means that p writes to variables u, v, and q reads from them). For every process $p \in \mathcal{P}$, we denote with $O(p) = \bigcup_{q \in \mathcal{P}} E(p,q)$ the set of *output* variables of p, and with $I(p) = \bigcup_{q \in \mathcal{P}} E(q,p)$ the set of *input* variables of p. We require that for all $p, q \in \mathcal{P}$: $O(p) \cap O(q) = \emptyset$, i.e., no two processes write to the same variable. Finally, we will denote with $\mathcal{P}^- = \mathcal{P} \setminus \{p_e\}$.

An architecture describes a distributed reactive system, with the environment providing the inputs via $O(p_e)$, and the system responding via $I(p_e)$. The pair $(\mathcal{P}, E)$ describes the architecture of the system as a multigraph, with $\mathcal{P}$ being the set of nodes, and $E(p,q)$ the set of directed $p \to q$ edges with the corresponding variables as labels.

Trees. We define a (full) *B-tree* T over some finite set B as the set of all nodes $x \in \left(2^B\right)^*$. A (possibly infinite) sequence of nodes $\pi = (x_1, x_2 \dots)$ forms a *path* in T, if for every $i \geq 1$ we have $x_{i+1} = x_i z$, for some $z \in 2^B$. For such a path π, we will use $\pi[i]$ to denote the element of π at the i-th position, while $\pi[i, \infty]$ denotes the infinite suffix of π starting at position i. An *A-labeled B-tree* T_λ is a B-tree equipped with a labeling function of its nodes, $\lambda : \left(2^B\right)^* \to 2^A$. For every node $x = yz \in T_\lambda$ with $z \in 2^B$ we denote with $\ell_\lambda(x) = z \cup \lambda(x)$, i.e., the ℓ_λ of x consists of the branch z from the parent and the label $\lambda(x)$. For a (possibly infinite) path $\pi = (x_1, x_2, \dots)$, we define with $\ell_\lambda(\pi) = (\ell_\lambda(x_1), \ell_\lambda(x_2) \dots)$.

Local strategies. For every process $p \in \mathcal{P}^-$, a *local strategy* σ_p is a function $\sigma_p : \left(2^{I(p)}\right)^* \to 2^{O(p)}$, setting the output variables of p according to the history of its input variables. Observe that every such local strategy σ_p can be viewed as a labeling of an $O(p)$-labeled $I(p)$-tree T_{σ_p}. A local strategy σ_p has *finite memory* if there exists a finite set $\mathcal{M}$, $m_0 \in \mathcal{M}$, and functions $f : \mathcal{M} \times 2^{I(p)} \to \mathcal{M}$ and $g : \mathcal{M} \to 2^{O(p)}$ such that for all $x = x_1 x_2 \dots x_k$ with $x_i \in 2^{I(p)}$, we have $\sigma_p(x) = g(f(\dots(f(f(m_0, x_1), x_2) \dots, x_k))$. The *memory* of σ_p is said to be $|\mathcal{M}|$, while if $|\mathcal{M}| = 1$, then σ_p is called *memoryless*.

Collective strategies. The *collective strategy* of the architecture $\mathcal{A}$ is a function $\sigma : \left(2^{O(p_e)}\right)^* \to 2^{V \setminus O(p_e)}$, mapping every finite sequence of the outputs of the environment to a subset of the outputs of the processes p according to the composition $(\sigma_p : p \in \mathcal{P}^-)$. The collective strategy σ can be viewed as a $(V \setminus O(p_e))$-labeled $O(p_e)$-tree T_σ and for any infinite path π in T_σ, we will call $\ell_\sigma(\pi)$ a *computation*. Hence, T_σ describes a distributed algorithm, and every infinite path $\pi = (x_1, x_2, \dots)$ starting from the root represents a distributed computation $\ell_\sigma(\pi)$, according to the local strategies $(\sigma_p : p \in \mathcal{P}^-)$.

Synthesis (realizability). We will consider distributed reactive systems with specifications given by temporal logic formulae. For temporal logic formulae we will consider LTL; see [5] for the formal syntax and semantics of LTL. The problem of *realizability* of a temporal logic formula ϕ in an architecture $\mathcal{A}$ asks whether there exist local strategies σ_p for every process p, such that for every infinite path π of the $(V \setminus O(p_e))$-labeled $O(p_e)$-tree T_σ of the collective strategy σ, with π starting from the root, we have $\ell_\sigma(\pi) \models \phi$. If ϕ admits such strategies σ_p for every $p \in \mathcal{P}^-$, then it is called *realizable*, and the collective strategy σ gives an *implementation* for ϕ on $\mathcal{A}$.

III. Synthesis for Reachability Specifications

In the current section we discuss the synthesis problem for reachability specifications, where the objective consists of propositional formulae connected with Boolean operators and non-nested $\mathcal{X}$ (next) operators. We will show that even under such restrictions, the synthesis problem remains undecidable for all architectures containing an information fork, via a reduction from the halting problem of Turing machines.

Fragment LTL$_\Diamond$. We consider LTL$_\Diamond$ that consists of formulae ϕ from the following LTL fragment:

$$\theta = P \mid \mathcal{X}P$$
$$\psi = \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2 \mid \neg\theta$$
$$\phi = Q \to \Diamond\psi$$

where P, Q are propositional formulae, $\mathcal{X}$ is the next operator, $\Diamond$ is the eventually temporal operator. We consider the standard semantics of LTL. Formula $\Diamond\psi$ represents a reachability objective, and Q will capture the initial input in the architecture.

Turing machines. Let M be a deterministic Turing machine fixed throughout this section and let $\mathcal{Q}$ be the set of states

of M (see [17] for detailed descriptions of Turing machines). The machine M works over the alphabet $\{0, 1, \sqcup\}$, and its tape is bounded by $\#$ symbols. The machine M cannot move left on a $\#$ symbol, and moving right to a $\#$ symbol effects in extending the tape by a blank symbol $\sqcup$. In our analysis, M starts with the empty tape. A *configuration* of M is a word $\#v\mathbf{q}au\sqcup\#$, where $a \in \{0, 1\}$, $v, u \in \{0, 1\}^*$ and $\mathbf{q} \in \mathcal{Q}$. Such a configuration has the standard interpretation as an infinite tape such that v is the part of the tape preceding the head, $\mathbf{q}$ is the current state of M, a is the letter under the head, and u is a sequence of symbols succeeding the head. The blank symbol $\sqcup$ represents the rightmost cell of the tape that has not been altered by M. We define the projection $\pi_\perp$ over words w from some alphabet containing $\perp$, such that $\pi_\perp(w)$ is the result of omitting all $\perp$ symbols from w. We define a *scattered configuration* C of M as a word over $\Sigma = \{0, 1, \sqcup, \perp, \#\} \cup \mathcal{Q}$ such that $\pi_\perp(C)$ is a configuration of M.

Information-fork architecture. We first consider the architecture A_0 (Figure 1), characterized as an *information fork* in [10], for which the problem of realizability has been shown to be undecidable, using LTL formulae with nested until operators (in [4]). Here we show that the problem remains undecidable for A_0 and specifications in the restricted fragment of LTL$_\Diamond$. This is obtained through a reduction from the halting problem of M, by constructing a specification $\phi \in$ LTL$_\Diamond$ which is realizable iff M halts on the empty input.

Proof idea. The architecture A_0 consists of the environment p_e and two processes p_1 and p_2. The processes act as I/O streams, outputting configurations of M; the environment sends separately to each process *next* and *stall* signals, indicating that the corresponding process should output the next letter from $\{0, 1, \sqcup, \#\} \cup \mathcal{Q}$ of the current configuration of M, or it should output $\perp$.

Construction of φ. First, we will provide a regular safety property φ which specifies that if the environment satisfies an *alternation* assumption, i.e., every stall signal is followed by a next signal, then p_1 and p_2 conform with a series of guarantees. The property φ does not belong to the LTL$_\Diamond$ fragment, but we will show how it can be expressed by a safety automaton A_{safe}. Then, we will prove that if φ is realizable, and the environment conforms with the alternation assumption, then the processes output a legal sequence of configurations of M, scattered with the $\perp$ symbol.

Conversion to LTL$_\Diamond$. Next, we will provide the specification for the synthesis problem $\phi \in$ LTL$_\Diamond$, such that ϕ is realizable iff φ is realizable and M halts on the empty input. Formula ϕ does not express φ directly, but it asserts that the environment simulates a run of A_{safe} faithfully, and finally one of the processes outputs a halting configuration of M. More precisely, the environment simulates a run of A_{safe} storing the current state of A_{safe} in a set of hidden variables $\{q_1, \ldots, q_m\} \in E(p_e, p_e)$, and ϕ encodes that eventually either (i) the environment cheats in the simulation of A_{safe}, or (ii) one of the processes outputs a halting state $\mathbf{q}$ of M, while the current state of A_{safe} is not rejecting (i.e., $\mathbf{q}$ was reached legally with respect to M). We will conclude that ϕ is realizable iff M halts on the empty input.

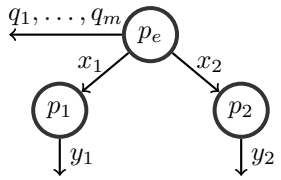


Fig. 1: The architecture A_0 which consists an information fork.

Formal proof. A safety automaton cannot express a scattered configuration that is finite. Thus, we define a *scattered preconfiguration* C (of M) as a (possibly infinite) word whose every finite prefix can be extended to a scattered configuration of M. A scattered preconfiguration is formally defined as a finite or infinite word over Σ that begins with $\#$, there is at most one symbol from $\mathcal{Q}$, there are no symbols after the second $\#$ and the $\sqcup$ symbol is followed by the $\#$ symbol.

Let C_1, C_2 be scattered preconfigurations. We denote with $\perp(C)$ the set of positions in C where $\perp$ occurs, and write $C_1 \parallel C_2$ if the symmetric difference of $\perp(C_1)$ and $\perp(C_2)$ has at most one element, i.e., $|\perp(C_1) \triangle \perp(C_2)| \leq 1$. We define as $C_1 \vdash C_2$ if $C_1 \parallel C_2$ and

(i) $\pi_\perp(C_2)$ follows legally from $\pi_\perp(C_1)$ according to M, or

(ii) both C_1, C_2 are infinite preconfigurations such that every finite prefix can be extended to finite preconfigurations C_1', C_2' such that $\pi_\perp(C_2')$ follows legally from $\pi_\perp(C_1')$.

For infinite words w_1, w_2, we define $w_1 \otimes w_2$ as a word over $\Sigma \times \Sigma$ such that the i-th letter of $w_1 \otimes w_2$ is a pair of the i-th letters of w_1, w_2. Observe that there are safety automata working over $\Sigma \times \Sigma$ that recognize the languages $\{C_1 \otimes C_2 : C_1 \parallel C_2\}$ and $\{C_1 \otimes C_2 : C_1 \vdash C_2\}$.

Construction of φ. We first construct the regular safety property $\varphi = \mathcal{L} \rightarrow \bigwedge_{0 \leq i \leq 4} Cond_i$, where $\mathcal{L}$ (the alternation assumption) and $Cond_i$ are defined as follows:

$\mathcal{L}$: for every process, every *stall* signal is followed by a *next* signal.

$Cond_0$: each process outputs $\perp$ when its input is *stall*, otherwise it outputs a letter from $\Sigma \setminus \{\perp\}$,

$Cond_1$: each process produces a sequence of scattered preconfigurations,

$Cond_2$: initially, each process produces two scattered configurations of M, whose projections are the first two valid configurations of M,

$Cond_3$: if starting from some position i, p_1 outputs consecutively C_1, C_2 and p_2 outputs consecutively C_1', C_2', then $C_1' \vdash C_1$ implies $C_2' \vdash C_2$ or $C_2' \nparallel C_2$,

$Cond_4$: if D, D' are outputs of p_1, p_2 up to some positions such that $D \parallel D'$ and $|\pi_\perp(D)| = |\pi_\perp(D')|$, then $\pi_\perp(D) = \pi_\perp(D')$.

We provide a high-level description of the construction of an alternating safety automaton A_{safe} (see [18] for the definition of alternating automata) which verifies that every execution satisfies φ. Note that A_{safe} can be transformed to a nondeterministic automaton by a standard power-set construction. Clearly, conditions $\mathcal{L}$, $Cond_0$ and $Cond_1$ can be expressed by a safety automaton. For the condition $Cond_2$, observe

that the first two configurations of M have at most 9 letters $\#\mathbf{q_0} \sqcup \#\#\mathbf{q_1}a \sqcup \#$, with $a \in \{0, 1, \epsilon\}$. To show that the rest of conditions can be expressed by a safety automaton, we assume that $\mathcal{L}$ is satisfied; otherwise those conditions do not have to be checked (note that if $\mathcal{L}$ is violated, A_{safe} accepts unconditionally). Because of $\mathcal{L}$, A_{safe} can verify that p_1 and p_2 conform with $Cond_2$ by checking the first 18 output letters. For the condition $Cond_3$, A_{safe} operates as follows: whenever it encounters a # symbol marking the beginning of a configuration, it splits universally. One copy looks for the next configuration, and the second copy, denoted by $\mathcal{A}_3$, verifies that $Cond_3$ holds at the current position, as follows. It ignores $\perp$ symbols and compares whether $C_1 \parallel C_1'$, configurations $\pi_\perp(C_1)$ and $\pi_\perp(C_1')$ are equal everywhere except for positions adjunct to the head of M, and the letters adjunct to the head are consistent with the transition of M. If one of these conditions is violated, $C_1' \nvdash C_1$, therefore $\mathcal{A}_3$ accepts the word regardless of what follows. Otherwise, if those conditions hold, i.e., $C_1' \vdash C_1$, $\mathcal{A}_3$ non-deterministically verifies one of the following conditions: $C_2' \nparallel C_2$ or $C_2' \vdash C_2$. Both conditions can be verified by safety automata, since C_2 and C_2' either start concurrently, or C_2 is delayed by 1 step from C_2'. For the condition $Cond_4$ observe that if $D \parallel D'$ and $|\pi_\perp(D)| = |\pi_\perp(D')|$, then $||D| - |D'|| \leq 1$ and the automaton needs to remember at most one symbol to compare $\pi_\perp(D)$ and $\pi_\perp(D')$. We can now prove the following lemma.

Lemma 1. *If φ is realizable, then for every $k \in \mathcal{N}$, in all executions where $\mathcal{L}$ holds, both p_1 and p_2 output sequences of scattered configurations whose $\pi_\perp$ projections are sequences of at least k consecutive valid configurations of M, starting with the initial configuration on the empty input.*

Proof: First note that there exist executions where the environment indeed satisfies $\mathcal{L}$, and thus p_1 and p_2 satisfy conditions $Cond_0$-$Cond_4$. The lemma clearly holds for $k = 1, 2$, due to conditions $Cond_0 - Cond_2$. For the inductive step, assume that the lemma holds for $k \geq 2$. Consider a sequence of inputs to p_1 consisting of *next* signals only. Then, there is a sequence of inputs to p_2 consisting of some number of *next* signals and exactly $|\pi_\perp(C_k)|$ *stall* signals placed in a such way that p_1 outputs $C_1 \dots C_k C_{k+1}$, p_2 outputs $C_1' \dots C_{k-1}' C_k'$, and $C_k C_{k+1}$, $C_{k-1}' C_k'$ are synchronized, i.e. they start at the same position and $C_k \parallel C_{k-1}'$, $C_{k+1} \parallel C_k'$. By the induction assumption $\pi_\perp(C_{k-1}')$ and $\pi_\perp(C_k) = \pi_\perp(C_k')$ are, respectively, $(k-1)$-th and k-th configurations of M. Therefore, $C_{k-1}' \vdash C_k$ and, by $Cond_3$, $C_k' \vdash C_{k+1}$. This implies that C_{k+1} is a finite scattered preconfiguration and $\pi_\perp(C_{k+1})$ is the $(k+1)$-th configuration of M.

Given that for an input consisting of *next* signals only, p_1 outputs $C_1 \dots C_k C_{k+1}$ satisfying the statement, we can show that regardless of the number of *stall* signals, under condition $\mathcal{L}$, p_1, p_2 output $k+1$ scattered configurations satisfying the statement. First, the condition $Cond_4$ implies that if p_2 also has an input sequence consisting of *next* signals alone, it will output the same sequence, that is, $C_1 \dots C_k C_{k+1}$. By a simple induction on the number of *stall* signals each process receives, and condition $Cond_4$, we conclude that for any number of *stall* signals, as long as $\mathcal{L}$ is satisfied by the environment, p_1, p_2 output $k+1$ scattered configurations whose projections are the first $k+1$ consecutive configurations of M. ■

Conversion to LTL$_\Diamond$. Given the safety automaton A_{safe} which verifies that φ is satisfied, we can construct a specification $\phi \in \text{LTL}_\Diamond$, such that ϕ is realizable if and only if the Turing machine M does not halt on the empty input. The environment uses the hidden (not visible to p_1, p_2) variables $q_1, \dots, q_k \in E(p_e, p_e)$ to simulate the automaton A_{safe}. We provide a high level description of the following formulae:

- Q specifies that the first state of A_{safe} according to the output variables $\{q_1, \dots q_m\}$ is compatible with the initial values of x_1, x_2, y_1 and y_2 (i.e. $\{q_1, \dots q_m\}$ represent the state of A_{safe} reached from the initial state after reading the initial values of x_1, x_2, y_1 and y_2; Q is propositional)
- ψ_1 specifies that A_{safe} has a transition from the current state to the next state, encoded by the values of $\{q_1, \dots q_m\}$ in the current and the next round, according to the value of variables x_1, x_2, y_1 and y_2 in the next round (i.e., p_e simulates A_{safe} faithfully; ψ_1 contains only propositionals and non-nested $\mathcal{X}$ operators).
- ψ_2 specifies that the current state of A_{safe} is not rejecting, and p_1 or p_2 outputs a halting state of M (i.e., some process reached a halting configuration of M, and both processes behaved according to A_{safe}; ψ_2 is propositional).

Finally, we construct $\phi = Q \rightarrow \Diamond(\neg\psi_1 \vee \psi_2)$, with $\phi \in \text{LTL}_\Diamond$. If ϕ is realizable, the processes satisfy ψ_2 in all runs where the environment faithfully simulates A_{safe} and conforms with condition $\mathcal{L}$(i.e., Q and ψ_1 are true). Then p_1, p_2 output a halting state of M and satisfy φ, which by Lemma 1, guarantees that the halting state was reached by a legal sequence of configurations of M. In the inverse direction, if M halts, then ϕ is realizable by (finite) local strategies which output a finite, legal sequence of configurations of M and conform with condition $Cond_0$. Hence, we obtain the following theorem.

Theorem 1. *The realizability of specifications from LTL$_\Diamond$ in A_0 is undecidable.*

Similarly as in [10], the above argument can be carried out to any architecture which contains an information fork, by introducing additional safety conditions in φ, which require that all processes propagate the inputs of the environment to the two processes constituting the information fork. It has also been shown in [10] that in architectures without information forks, the realizability of every LTL specification is decidable. Hence, Theorem 1 together with the results from [10] lead to the following corollary.

Corollary 1. *For every architecture $\mathcal{A}$, the realizability of specifications from LTL$_\Diamond$ in $\mathcal{A}$ is decidable iff $\mathcal{A}$ does not contain an information fork.*

IV. Synthesis for Safety Specifications

In the current section we consider safety specifications where the safety condition consists of propositional formulae connected with Boolean operators, and the $\mathcal{X}$ temporal operator. First, we show that the synthesis problem is undecidable

for architectures containing an information fork-meet (see Figure 3), by a similar construction as in the case of LTL$_\Diamond$. Then we show that the problem is decidable for a family of star architectures, despite the existence of information forks.

Fragment LTL$_\Box$. We consider LTL$_\Box$ that consists of formulae ϕ from the following LTL fragment:

$$\psi = P \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \neg\psi \mid \mathcal{X}\psi$$
$$\phi = Q \wedge \Box\psi$$

where P, Q are propositional formulae, and $\Box$ is the globally operator. We consider the standard semantics of LTL. The $\Box\psi$ part of ϕ specifies a safety condition, and we interpret Q as the initial conditions. The fragment LTL$_\Box$ can express safety specifications, one of the most basic specifications in verification.

While the information fork criterion is decisive for the undecidability of reachability specifications, here we extend this criterion to the family of star architectures of $n+1$ processes, denoted as S_n (i.e., p_e is the central process , and $\bigcup_i I(p_i) = O(p_e)$) (Figure 2) and show that: (i) the realizability of some $\phi \in$ LTL$_\Box$ in S_n is decidable if all processes receive pairwise disjoint inputs, (ii) it is undecidable if $n \geq 3$ and we allow overlapping inputs. The latter can be generalized to all architectures which contain such a structure, which we call an *information fork-meet*.

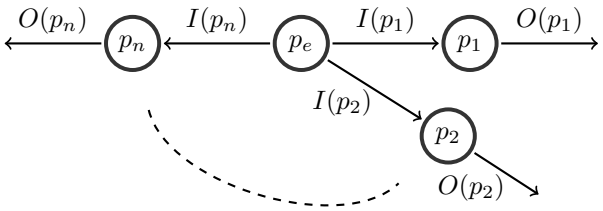


Fig. 2: The family of start architectures S_n.

A. Overlapping inputs

Here we demonstrate undecidability of realizability of specifications $\phi \in$ LTL$_\Box$ for star architectures with overlapping inputs, and with ϕ having $\mathcal{X}$-depth 1 (i.e., ϕ belongs to a subclass of LTL$_\Box$ where $\mathcal{X}$ operators are not nested). We first consider the star architecture A_1 (Figure 3), and obtain the undecidability of realizability of such specifications via a reduction from the (non) halting problem.

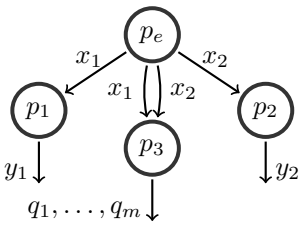


Fig. 3: The architecture A_1 consists an information fork-meet.

Given a Turing machine M, recall the specification φ (from Section 3 for LTL$_\Diamond$) encoding conditions $\mathcal{L}$ and $Cond_0 - Cond_4$ through the safety automaton A_{safe}. In contrast with the previous section, here we require that process p_3 (instead of p_e) faithfully simulates the safety automaton A_{safe} using the output variables $q_1, \ldots q_m \in E(p_3, p_e)$. Note that A_{safe} operates on the variables x_1, x_2, y_1, y_2, while p_3 does not have access to y_1 and y_2. However, it can infer these values by simulating p_1 and p_2 internally, since p_3 receives both x_1 and x_2 (overlapping inputs).

Formal proof. We provide a high level description of the following formulae:

- Q specifies that the first state of A_{safe} according to the output variables $\{q_1, \ldots q_m\}$ is compatible with the initial values of x_1, x_2, y_1 and y_2 (i.e. $\{q_1, \ldots q_m\}$ represent the state of A_{safe} reached from the initial state after reading the initial values of x_1, x_2, y_1 and y_2; Q is propositional)
- ψ_1 specifies that A_{safe} has a transition from the current state to the next state, encoded by the values of $\{q_1, \ldots q_m\}$ in the current and the next round, according to the value of variables x_1, x_2, y_1 and y_2 in the next round (i.e., p_e simulates A_{safe} faithfully; ψ_1 contains only propositionals and non-nested $\mathcal{X}$ operators).
- ψ_2 specifies that p_1 and p_2 do not output a halting state of M (i.e., M does not terminate; ψ_2 is propositional).
- ψ_3 specifies that A_{safe} does not reach a rejecting state (i.e., the processes conform to conditions $Cond_0$-$Cond_4$ or the environment violates $\mathcal{L}$; ψ_3 is propositional).

We construct $\phi = Q \wedge \Box(\psi_1 \wedge \psi_2 \wedge \psi_3)$. Similarly as in the case of LTL$_\Diamond$, if ϕ is realizable, p_3 faithfully simulates A_{safe} (Q and ψ_1 are true), and p_1, p_2 satisfy φ in all runs where the environment conforms with condition $\mathcal{L}$ (ψ_3 is true). By Lemma 1, p_1 and p_2 output a legal sequence of configurations of M, and ψ_2 guarantees that M does not halt. In the inverse direction, if M does not halt, ϕ is realizable by local strategies where (i) p_1, p_2 output a legal sequence of configurations of M and conform with condition $Cond_0$, and (ii) p_3 faithfully simulates A_{safe}. Hence we have the following result.

Theorem 2. *The realizability of specifications from LTL$_\Box$ in A_1 is undecidable.*

Remark 1. *We remark that our proof of undecidability in Theorem 2 makes use of infinite-memory strategies, since the processes p_1 and p_2 are required to output an infinite, non-halting computation. However, the realizability problem for LTL$_\Box$ in A_1 remains undecidable even if we restrict the strategies to be finite-memory. We refer to the longer version of this paper in [19] for the proof.*

Information fork-meet. We say that an architecture $\mathcal{A} = (\mathcal{P}, p_e, V, E)$ has an *information fork-meet* if there are three processes $p_1, p_2, p_3 \in \mathcal{P}^-$ and paths π_1, π_2 in the underlying graph such that

1) the first edges in π_1, π_2 are labeled by output variables of p_e,
2) the last edge of π_1 is an input variable of p_1, but not p_2
3) the last edge of π_2 is an input variable of p_2, but not p_1
4) the last edges of π_1, π_2 are input variables of p_3

Observe that an information fork-meet is a special case of information fork, with a third process that collects all information that is divided between p_1 and p_2.

As in the case of LTL$_\Diamond$, the undecidablity argument can be carried to any architecture containing such a structure, by introducing additional conditions in φ which require the rest of the processes to propagate the inputs of the environment to p_1, p_2 and p_3 accordingly.

Corollary 2. *The realizability of LTL$_\Box$ specifications in architectures containing an information fork-meet is undecidable.*

B. Pairwise disjoint inputs

In this subsection we discuss synthesis for formulae $\phi \in$ LTL$_\Box$ for the class of star architectures, with the additional property that all pairs of processes receive disjoint inputs (i.e., $\forall i \neq j : I(p_i) \cap I(p_j) = \emptyset$), denoted as $\overline{S}_n$. Our goal is to prove decidability of realizability of such $\phi \in$ LTL$_\Box$ in every architecture $\mathcal{A} \in \overline{S}_n$, by showing that whenever such ϕ is realizable, it admits strategies of bounded memory.

Consider some architecture $\mathcal{A} \in \overline{S}_n$ and an arbitrary $\phi = Q \wedge \Box\psi \in$ LTL$_\Box$, with the nesting level of $\mathcal{X}$ operators in ψ being k. Assume that ϕ is realizable in $\mathcal{A}$ by local strategies σ_i for every process p_i. These strategies can be represented by $O(p_i)$-labeled $I(p_i)$-trees T_{σ_i}. We will show how to construct strategies τ_i that also realize ϕ, where each tree $I(p_i)$-tree T_{τ_i} representing τ_i is defined from first $2^{2^k|V|}+1$ levels of T_{σ_i} by applying a *folding function* given below. We first define the notion of some $i \in \mathcal{N}$ *closing* $\neg\psi$ in some computation.

Definition 1. *For a computation $\ell(\pi)$ and some $i \in \mathcal{N}$ we say that i closes $\neg\psi$ in $\ell(\pi)$ if $\ell(\pi)[i-k,\infty] \models \neg\psi$.*

Remark 2. *$\ell(\pi) \models \Box\psi$ iff no i closes $\neg\psi$ in $\ell(\pi)$.*

Let $\sigma_1, \ldots, \sigma_n$ be local strategies and σ be the collective strategy induced by $\sigma_1, \ldots, \sigma_n$. For every $i \in \{1, \ldots, n\}$, the local strategy σ_i is represented by an $O(p_i)$-labeled $I(p_i)$-tree T_{σ_i}. For every node $x \in T_{\sigma_i}$, with $|x| \geq k$, we denote with $\overline{\pi}_x = (x_k, x_{k-1} \ldots x_1)$ the k-node suffix of the unique path to $x = x_1$, and define the *type* of x under σ_i as $t_{\sigma_i}(x) = \ell_{\sigma_i}(\overline{\pi}_x)$. For every level $l \geq k$ we define the type of l under σ as $t_\sigma(l) = \{t_{\sigma_i}(x) : i \in \{1, \ldots, n\}, x \in T_{\sigma_i} \text{ and } |x| = l\}$, i.e., the type of a level l is the set of the types of the nodes of level l of every T_{σ_i}, where $i \in \{1, \ldots, n\}$. Note that there exist at most $2^{k|V|}$ distinct types of nodes. Consequently, there exist at most $2^{2^{k|V|}}$ distinct types of levels.

We naturally extend the definition of types to nodes of the $(V \setminus O(p_e))$-labeled $O(p_e)$-tree T_σ as $t_\sigma(x) = \ell_\sigma(\overline{\pi}_x)$. Consider some computation $\ell_\sigma(\pi)$ in T_σ. Observe that whether some i closes $\neg\psi$ in π depends only on the $\ell_\sigma(\pi)[i]$ i.e., the type $t_\sigma(\pi[i])$ determines whether i closes $\neg\psi$ in π. Hence, we have the following remark:

Remark 3. *For a formula $\phi \in$ LTL$_\Box$ there exists a set of types Δ such that for every tree T_σ, a path π in T_σ satisfies ϕ if $\ell_\sigma(\pi)[1] \models Q$ and for all $i \in \mathcal{N}$, we have $t_\sigma(\pi[i]) \in \Delta$, i.e., the set of types of nodes in T_σ is a subset of Δ.*

Folding function. Assume that there exist two levels $l_1 < l_2$ such that $t_\sigma(l_1) = t_\sigma(l_2)$. Then for every tree T_{σ_i}, for every node x in level l_2 there exists a node y in level l_1 such that $t_{\sigma_i}(x) = t_{\sigma_i}(y)$, i.e., x and y have the same type. For such l_1, l_2, and every process p_i, we define the folding function $f_i : \left(2^{I(p_i)}\right)^* \to \left(2^{I(p_i)}\right)^*$ recursively as follows:

$$f_i(x) = \begin{cases} x \text{ if } |x| < l_2 \\ y \text{ if } |x| = l_2 \text{ where } |y| = l_1 \text{ and } t_{\sigma_i}(x) = t_{\sigma_i}(y) \\ f_i(f_i(y)z) \text{ if } |x| > l_2 \text{ for } x = yz \text{ with } z \in 2^{I(p)} \end{cases}$$

and construct local strategies $\tau_i(x) = \sigma_i(f_i(x))$. Hence, every strategy τ_i behaves as σ_i up to level l_2, while for nodes further below, it maps them to nodes between levels l_1 and l_2, by recursively folding the levels l_1 and l_2 with respect to the types of their nodes. Since the collective strategies σ and τ behave identically on the first l_1 levels, τ realizes the propositional Q. The following analysis focuses on the $\Box\psi$ part of ϕ.

The strategies τ_i preserve the types under σ_i of all local nodes up to level l_2, and only those. Because of the pairwise disjoint inputs, this property is implied for the global nodes of the collective strategy τ as well. The set of all such types serves as the set Δ of Remark 3, which in turn guarantees that the collective strategy τ also realizes ϕ, as it does not introduce new types. We formalize these arguments below.

The following lemma establishes that for all nodes x in all T_{τ_i}, the type of x is the same as the type of its image under f_i in the corresponding T_{σ_i}.

Lemma 2. *For every $x \in \left(2^{I(p_i)}\right)^*$ with $|x| \geq k$, we have that $t_{\tau_i}(x) = t_{\sigma_i}(f_i(x))$.*

Proof: Our proof proceeds by induction on $|x|$:

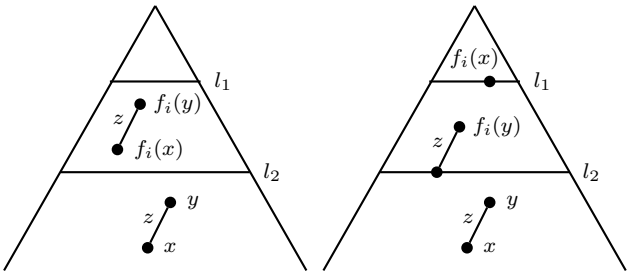
1) $|x| < l_2$: For all nodes w in $\overline{\pi}_x$, we have that $\tau_i(w) = \sigma_i(f_i(w)) = \sigma_i(w)$, hence $\ell_{\tau_i}(\overline{\pi}_x) = \ell_{\sigma_i}(\overline{\pi}_x)$ and thus $t_{\tau_i}(x) = t_{\sigma_i}(f(x))$.
2) $|x| = l_2$: The statement holds by definition.
3) $|x| = m+1$: Let $x = yz$ with $|y| = m$. By the inductive hypothesis, $t_{\tau_i}(y) = t_{\sigma_i}(f_i(y))$. We distinguish between the following cases, depending on whether $f_i(y)$ extended by z hits the level l_2 (Figure 4):
   (i) $|f_i(y)| < l_2 - 1$: Then $f_i(x) = f_i(f_i(y)z) = f_i(y)z$, that is, if we reach node x by extending node y by an edge z, the same holds for their corresponding images under f_i. Then $\tau_i(x) = \sigma_i(f_i(x)) = \sigma_i(f_i(y)z)$, thus $t_{\tau_i}(x) = t_{\sigma_i}(f_i(y)z) = t_{\sigma_i}(f_i(x))$ (i.e., the strategy τ_i will label x as σ_i labels its image $f_i(x)$, and the types of these two nodes are equal).
   (ii) $|f_i(y)| = l_2 - 1$: By construction, $t_{\sigma_i}(f_i(x)) = t_{\sigma_i}(f_i(y)z)$ (i.e., $f_i(y)$ extended by z hits level l_2, and the folding function f_i will bring x to level l_1, to a node of the same type). Then $\tau_i(x) = \sigma_i(f_i(x)) = \sigma_i(f_i(y)z)$, hence as in (i), $t_{\tau_i}(x) = t_{\sigma_i}(f_i(y)z) = t_{\sigma_i}(x)$.

The desired result follows. ∎

The following remark observes that for every architecture from $\overline{S}_n$, every node in the collective strategy tree corresponds to a unique set of nodes in the local strategy trees and vice versa, and that the collective strategy on that node equals the union of the local strategies on the corresponding local nodes.

Remark 4. *The following assertions hold:*

1) *For every global node $x = x^1 x^2 \ldots x^m$ in T_σ with every $x^i \in 2^{O(p_e)}$, for every tree T_{σ_j}, there exists a (unique) node $x_j = x_j^1 x_j^2 \ldots x_j^m$ such that $x_j^i = x^i \cap 2^{I(p_j)}$, and*



(a) Case (i) (b) Case (ii)

Fig. 4: The two cases of the inductive step of Lemma 2.

2) for every set of nodes $\{x_j = x_j^1 x_j^2 \dots x_j^m\}$ with one x_j from each T_{σ_j}, there exists a (unique) global node x such that for all i we have $x^i = \bigcup_j x_j^i$.

Moreover, for every collective strategy σ, we have $\sigma(x) = \bigcup_j \sigma_j(x_j)$.

It follows from the above remark and Lemma 2, that for every $x \in T_\sigma$ we have that $t_\tau(x) = t_\sigma(f(x))$, where $f(x) = \bigcup_i f_i(x_i)$. That is, the local folding functions f_i result in a unique, global folding function f, and the types in the corresponding collective strategy tree are preserved between the global nodes, and their images under f. This implies that the set of types occurring in T_τ is a subset of types of T_σ. Then, by Remark 3 we conclude:

Lemma 3. *The collective strategy τ implements ϕ.*

Hence, whenever for a realizable $\phi \in \text{LTL}_\Box$ exist levels l_1 and l_2 with the same type under σ, we can construct a collective strategy τ for which every local strategy τ_i uses only the first l_2 levels of the corresponding σ_i, and Lemma 3 guarantees that τ implements ϕ. By our previous observation and the pigeonhole principle, l_2 is upper bounded by $2^{2^{k|V|}}+1$, and thus every local strategy τ_i operates in the first $2^{2^{k|V|}}+1$ levels of the corresponding $I(p_i)$-tree. There are a bounded number of local strategies with this property, thus the problem of realizability in this case reduces to exhaustively exploring all of them. Moreover, it follows from our analysis that local nodes in the same level and having the same type can be merged, since the local strategy that behaves identically in both subtrees preserves the set of types appearing in the global tree. Hence, the width of each level is bounded by the number of different possible types, $2^{k|V|}$. This leads to Theorem 3 (we refer to [19] for the formal proof).

Theorem 3. *The realizability of $\phi \in$ LTL$_\Box$ for the class $\overline{S}_n$ of star architectures with pairwise disjoint inputs is decidable in EXPSPACE.*

V. Synthesis without the next operator

In the current section we consider a fragment of LTL without the $\mathcal{X}$ operator, for which the problem of realizability is decidable in non-deterministic exponential time in the size of the specification.

Fragment LTL$_{\text{AG}}$. We consider LTL$_{\text{AG}}$ that consists of formulae ϕ from the following LTL fragment:

$$\begin{aligned}\phi &= \bigwedge_i \Box P_i \rightarrow \left(\bigwedge_i \Box Q_i \wedge \bigwedge_i \Box\Diamond R_i \wedge \bigwedge_i \Diamond F_i \right) \\ &\equiv \Box \bigwedge_i P_i \rightarrow \left(\Box \bigwedge_i Q_i \wedge \bigwedge_i \Box\Diamond R_i \wedge \bigwedge_i \Diamond F_i \right) \\ &\equiv \Box P \rightarrow \left(\Box Q \wedge \bigwedge_i \Box\Diamond R_i \wedge \bigwedge_i \Diamond F_i \right)\end{aligned}$$

for $i \in \{1, \dots m\}$, with P_i, Q_i, R_i, F_i propositional formulae, and $P = \bigwedge_i P_i$, $Q = \bigwedge_i Q_i$. We consider the standard semantics of LTL. The LTL$_{\text{AG}}$ can express specifications that consist of conjunction of safety assumptions, and guarantees where each guarantee is a safety, reachability, or a liveness condition.

A propositional formula Q has the property that can either be realized in a single step, or is not realizable. This implies that realizable formulae $\Box Q$ admit memoryless strategies which repeat the single step realization of Q. A similar argument establishes that reachability and safety specifications of propositional formulae are equivalent with respect to realizability. We formally state these observations in Lemmas 4 and 5, and refer to [19] for the proofs.

Lemma 4. *Let $\mathcal{A}$ be any architecture. Every formula $\psi = \Box Q$, for some propositional Q, is realizable in $\mathcal{A}$ iff it is realizable by memoryless strategies.*

Lemma 5. *Let $\mathcal{A}$ be any architecture. For every formula $\psi = \Box Q$ for some propositional Q, ψ is realizable in $\mathcal{A}$ iff $\psi' = \Diamond Q$ is realizable in $\mathcal{A}$.*

Lemma 6 shows that the realizability of some $\phi \in$ LTL$_{\text{AG}}$ reduces to realizing a set of safety formulae of the form of Lemma 4.

Lemma 6. *Let $\mathcal{A}$ be any architecture and $\phi = \Box P \rightarrow (\Box Q \wedge \bigwedge_i \Box\Diamond R_i \wedge \bigwedge_i \Diamond F_i) \in$ LTL$_{\text{AG}}$. The formula ϕ is realizable in $\mathcal{A}$ iff every $\phi_{R_i} = \Box(P \rightarrow (Q \wedge R_i))$ and every $\phi_{F_i} = \Box(P \rightarrow (Q \wedge F_i))$ is realizable in $\mathcal{A}$.*

Proof: (i) For the right to left direction, assume that there exist families of memoryless (by Lemma 4) local strategies $(\sigma_j^{R_i})$ and $(\sigma_j^{F_i})$ for every process p_j, such that the collective strategy σ^{R_i} implements ϕ_{R_i}, and the collective strategy σ^{F_i} implements ϕ_{F_i}. Construct local strategies τ_j such that for every $x = yz$ with $|z| = (1 + |x| \mod 2m)$, we have $\tau_j(x) = \sigma_j^{R_{|z|}}(z)$ if $|z| \leq m$, and $\tau_j(x) = \sigma_j^{F_{|z|-m}}(z)$ if $|z| > m$ (i.e. the local strategy τ_j repeatedly alternates between all the strategies $\sigma_j^{R_i}$ in the first m steps, and between all the strategies $\sigma_j^{F_i}$ the next m steps). Let τ be the collective strategy of all τ_j and consider an arbitrary path π in T. Either $\ell_\tau(\pi)[k] \models \neg P$ for some k, or for all k, it holds $\ell_\tau(\pi)[k] \models P$, and by construction, for $i = 1 + k \mod 2m$, we have $\ell_\tau(\pi)[k] \models Q \wedge R_i$ when $i \leq m$ and $\ell_\tau(\pi)[k] \models Q \wedge F_{i-m}$ when $i > m$. In both cases, $\ell_\tau(\pi) \models \phi$.
(ii) For the left to right direction, assume that for some i, ϕ_{R_i} is not realizable (the analysis is similar for ϕ_{F_i}). By Lemma 5, $\Diamond(P \rightarrow (Q \wedge R_i)$ is not realizable. Hence, for any collective strategy σ there exists some path π in T_σ, such that for all

k, we have $\ell_\sigma(\pi)[k] \models P \wedge (\neg Q \vee \neg R_i)$, and σ does not implement ϕ. ∎

Hence, Lemma 6 establishes that every formula $\phi \in \text{LTL}_{\text{AG}}$ is realizable if and only if it admits local strategies for all the corresponding ϕ_{F_i}, ϕ_{R_i}, by providing a constructive argument. As a consequence of Lemma 4, deciding whether every ϕ_{F_i}, ϕ_{R_i} is realizable reduces to realizing the propositional formulae $(P \rightarrow (Q \wedge R_i)$ and $(P \rightarrow (Q \wedge F_i)$. This can be done in NEXPTIME, by having a non-deterministic Turing machine guessing the local strategies of all processes, and verifying that such strategies satisfy the formula under all the (exponentially many) possible inputs of the environment. We show that the problem is also NEXPTIME-hard, via a reduction from the Dependency Quantifier Boolean Formula (DQBF) validity problem introduced in [20] to study time bounded multi-player alternating machines. A DQBF is a quantified Boolean formula with a succinct description of dependencies between the quantified variables. Every DQBF has an equivalent form in which all existentially quantified variables are substituted by existentially quantified Skolem functions defined over their dependencies, and appearing at the beginning of the formula (e.g. $\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) \varphi(x_1, x_2, y_1, y_2)$ is a DQBF stating that y_i depends on x_i, and has a functional form $\exists \sigma_1 \exists \sigma_2 \forall x_1 \forall x_2 \varphi(x_1, x_2, \sigma_1(x_1), \sigma_2(x_2))$ with σ_1, σ_2 the Skolem functions).

Lemma 7. *Given an architecture $\mathcal{A}$ and a formula $\phi \in$ LTL$_{AG}$, deciding whether ϕ is realizable in $\mathcal{A}$ is NEXPTIME-hard.*

Proof: Consider any DQBF formula $\psi : \forall x_1 \ldots \forall x_k \exists y_1(\overrightarrow{x_1}) \ldots \exists y_n(\overrightarrow{x_n}) \varphi(x_1, \ldots x_k, y_1 \ldots y_n)$ with k universally quantified variables x_i and n existentially quantified variables y_i. We assume w.l.o.g. that the dependencies of each y_i are only on some universally quantified variables $\overrightarrow{x_i}$. We construct the architecture $\mathcal{A} = (\mathcal{P}, p_e, V, E)$, where $\mathcal{P}$ contains $n+1$ processes, $V = \{x_i \in \psi\} \cup \{y_i \in \psi\}$, process p_i receives as inputs from the environment all $\overrightarrow{x}_i$, outputs variable y_i, while the environment uses all remaining x_j as hidden variables. We construct the specification $\phi = \Box\varphi \in \text{LTL}_{\text{AG}}$. Both $\mathcal{A}$ and ϕ are polynomial in the size of ψ. Because of Lemma 4, ϕ is realizable in $\mathcal{A}$ iff φ is realizable in $\mathcal{A}$. In turn, φ is realizable iff ψ is valid, with local strategies σ_i corresponding to the Skolem functions in the functional form of ψ, and universal variables corresponding to all possible choices of the environment in $\mathcal{A}$. Since DQBF validity is NEXPTIME-hard [20], the statement follows. ∎

Hence, we have the following result.

Theorem 4. *Given an architecture $\mathcal{A}$ and a specification $\phi \in$ LTL$_{AG}$, the realizability of ϕ in $\mathcal{A}$ is NEXPTIME-complete.*

Observe that Lemma 6 reduces the problem of realizability of some $\varphi \in \text{LTL}_{\text{AG}}$ to realizing a set of formulae of the form $\Box Q$, where Q is propositional. This in turn is reducible to DQBF validity (because of Lemma 4), and because of Lemma 7, the two problems are equivalent. In consequence, efficient algorithms for solving DQBF, such as [21], yield efficient synthesis procedures for LTL_{AG}, and vice versa. Moreover, if the DQBF tool outputs the corresponding Skolem functions, then a witness collective strategy for realizability can be obtained.

VI. Conclusions

In this paper we studied the distributed synthesis problem for relevant fragments of LTL. We presented a much finer characterization of undecidability results for distributed synthesis in terms of LTL fragments that uses eventually, globally and next operators. In contrast to previous decidability results that were non-elementary, we identify fragments where the complexity is EXPSPACE (or NEXPTIME-complete). An interesting direction of future work would be to develop algorithms for the problems for which we establish decidability, obtain efficient implementations of the algorithms for distributed synthesis problems, and finally consider some case-studies of practical examples.

Acknowledgments. The research was supported by Austrian Science Fund (FWF) Grant No P 23499- N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), Microsoft faculty fellows award, the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering), the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

References

[1] A. Church, "Logic, arithmetic and automata," in *Proceedings of the international congress of mathematicians*, pp. 23–35, 1962.
[2] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.
[3] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," POPL '89, pp. 179–190, ACM, 1989.
[4] A. Pnueli and R. Rosner, "Distributed reactive systems are hard to synthesize," SFCS '90, pp. 746–757 vol.2, 1990.
[5] A. Pnueli, "The temporal logic of programs," in *FOCS*, pp. 46–57, 1977.
[6] J. H. Reif, "Universal games of incomplete information," STOC '79, pp. 288–308, ACM, 1979.
[7] G. L. Peterson and J. H. Reif, "Multiple-person alternation," in *FOCS*, pp. 348–363, 1979.
[8] O. Kupferman and M. Y. Vardi, "Synthesizing distributed systems," in *LICS*, pp. 389–398, 2001.
[9] S. Mohalik and I. Walukiewicz, "Distributed games," in *FSTTCS*, pp. 338–351, 2003.
[10] B. Finkbeiner and S. Schewe, "Uniform distributed synthesis," LICS, pp. 321–330, 2005.
[11] R. Alur, S. La Torre, and P. Madhusudan, "Playing games with boxes and diamonds," in *CONCUR*, pp. 127–141, 2003.
[12] R. Alur and S. La Torre, "Deterministic generators and games for LTL fragments," *ACM Trans. Comput. Log.*, vol. 5, no. 1, pp. 1–25, 2004.
[13] J. Kretínský and J. Esparza, "Deterministic automata for the (F, G)-fragment of LTL," in *CAV*, pp. 7–22, 2012.
[14] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, LNCS 3855, Springer, pp. 364–380, 2006.
[15] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of AMBA AHB from formal specification: A case study," *STTT*, 2011.
[16] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Interactive presentation: Automatic hardware synthesis from specifications: a case study," in *DATE*, pp. 1188–1193, 2007.
[17] C. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994.
[18] O. Kupferman, M. Y. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," *Journal of the ACM (JACM)*, vol. 47, no. 2, pp. 312–360, 2000.
[19] K. Chatterjee, T. A. Henzinger, J. Otop, and A. Pavlogiannis, "Distributed synthesis for LTL fragments," 2013. Technical Report: IST-2013-128 https://repository.ist.ac.at/130/1/Distributed_Synthesis.pdf.
[20] G. Peterson, J. Reif, and S. Azhar, "Lower bounds for multiplayer non-cooperative games of incomplete information," *Journal of Computers and Mathematics with Applications*, vol. 41, pp. 957–992, 2001.
[21] A. Fröhlich, G. Kovásznai, and A. Biere, "A DPLL algorithm for solving DQBF," *Pragmatics of SAT*, vol. 2012, 2012.

Counter-Strategy Guided Refinement of GR(1) Temporal Logic Specifications

Rajeev Alur, Salar Moarref, and Ufuk Topcu
University of Pennsylvania, Philadelphia, USA. {alur,moarref,utopcu}@seas.upenn.edu

***Abstract*—The reactive synthesis problem is to find a finite-state controller that satisfies a given temporal-logic specification regardless of how its environment behaves. Developing a formal specification is a challenging and tedious task and initial specifications are often unrealizable. In many cases, the source of unrealizability is the lack of adequate assumptions on the environment of the system. In this paper, we consider the problem of automatically correcting an unrealizable specification given in the generalized reactivity (1) fragment of linear temporal logic by adding assumptions on the environment. When a temporal-logic specification is unrealizable, the synthesis algorithm computes a counter-strategy as a witness. Our algorithm then analyzes this counter-strategy and synthesizes a set of candidate environment assumptions that can be used to remove the counter-strategy from the environment's possible behaviors. We demonstrate the applicability of our approach with several case studies.**

I. Introduction

Automatically synthesizing a system from a high-level specification is an ambitious goal in the design of reactive systems. The synthesis problem is to find a system that satisfies the specification regardless of how its environment behaves. Therefore, it can be seen as a two-player game between the environment and the system. The environment attempts to violate the specification while the system tries to satisfy it. A specification is *unsatisfiable* if there is no input and output trace that satisfies the specification. A specification is *unrealizable* if there is no system that can implement the specification. That is, the environment can behave in such a way that no matter how the system reacts, the specification would be violated. In this paper we consider specifications which are satisfiable but unrealizable. We address the problem of strengthening the constraints over the environment by adding assumptions in order to achieve realizability.

Writing a correct and complete formal specification which conforms to the (informal) design intent is a hard and tedious task [4], [5]. Initial specifications are often incomplete and unrealizable. Unrealizability of the specification is often due to inadequate environment assumptions. In other words, assumptions about the environment are too weak, leading to an environment with too many behaviors that makes it impossible for the system to satisfy the specification. Usually there is only a rough and incomplete model of the environment in the design phase; thus it is easy to miss assumptions on the environment side. We would like to automatically find such *missing* assumptions that can be added to the specification and make it realizable. Computed assumptions can be used to give the user insight into the specification. They also provide ways to correct the specification. In the context of compositional synthesis [6], [9], derived assumptions based on the components specifications can be used to construct interface rules between the components.

An unrealizable specification cannot be executed or simulated which makes its debugging a challenging task. Counter-strategies are used to explain the reason for unrealizabilty of linear temporal logic (LTL) specifications [5]. Intuitively, a counter-strategy defines how the environment can react to the outputs of the system in order to enforce the system to violate the specification. Konighofer et al. in [5] show how such a counter-strategy can be computed for an unrealizable LTL specification. The requirement analysis tool RATSY [2] implements their method for a fragment of LTL known as generalized reactivity (1) (GR(1)). We also consider GR(1) specifications in this paper because the realizability and synthesis problems for GR(1) specifications can be solved efficiently in polynomial time and GR(1) is expressive enough to be used for interesting real-world problems [3], [12].

Counter-strategies can still be difficult to understand by the user especially for larger systems. We propose a debugging approach which uses the counter-strategies to strengthen the assumptions on the environment in order to make the specification realizable. For a given unrealizable specification, our algorithm analyzes the counter-strategy and synthesizes a set of *candidate* assumptions in the GR(1) form (see section II). Any of the computed candidate assumptions, if added to the specification, restricts the environment in such a way that it cannot behave according to the counter-strategy—without violating its assumptions—anymore. Thus we say the counter-strategy is ruled out from the environment's possible behaviors by adding the candidate assumption to the specification.

The main flow for finding the missing environment assumptions is as follows. If the specification is unrealizable, a counter-strategy is computed for it. A set of *patterns* are then synthesized by processing an abstraction of the counter-strategy. Patterns are LTL formulas of special form that define the structure for the candidate assumptions. We ask the user to specify a set of variables to be used for generating candidates for each pattern. The user can specify the set of variables which she thinks contribute to unrealizability or are underspecified. The variables are used along with patterns to generate the candidate assumptions. Any of the synthesized assumptions can be added to the specification to rule out the counter-strategy. The user can choose an assumption from the candidates in an interactive way or our algorithm can automatically search for it. The chosen assumption is then added to the specification

This research was partially supported by NSF Expedition in Computing project ExCAPE (grant CCF 1138996), and AFOSR (grant number FA9550-12-1-0302).

ISBN 978-0-9835678-3-7/13.

and the process is repeated with the new specification.

The contributions of this paper are as follows: We propose algorithms to synthesize environment assumptions by directly processing the counter-strategies. We give a counter-strategy guided synthesis approach that finds the missing environment assumptions. The suggested refinement can be validated by the user to ensure compatibility with her design intent and can be added to the specification to make it realizable. We demonstrate our approach with examples and case studies.

The problem of correcting an unrealizable LTL specification by constructing an additional environment assumption is studied by Chatterjee et al. in [4]. They give an algorithm for computing the assumption which only constrains the environment and is as weak as possible. Their approach is more general than ours as they consider general LTL specifications. However, the synthesized assumption is a Büchi automaton which might not translate to an LTL formula and can be difficult for the user to understand (for an example, see Fig. 3 in [4]). Moreover, the resulting specification is not necessarily compatible with the design intent [7]. Our approach generates a set of assumptions in GR(1) form that can easily be validated by the user and be used to make the specification realizable.

The closest work to ours is the work by Li et al. [7] where they propose a template-based specification mining approach to find additional assumptions on the environment that can be used to rule out the counter-strategy. A template is an LTL formula with at least one placeholder, $?_b$, that can be instantiated by the Boolean variable b or its negation. Templates are used to impose a particular structure on the form of generated candidates and are engineered by the user based on her knowledge of the environment. A set of candidate assumptions is generated by enumerating all possible instantiations of the defined templates. For a given counter-strategy, their method finds an assumption from the set of candidate assumptions which is satisfied by the counter-strategy. By adding the negation of such an assumption to the specification, they remove the behavior described by the counter-strategy from the environment. Similar to their work, we consider unrealizable GR(1) specifications and achieve realizability by adding environment assumptions to the specification. But, unlike them, we directly work on the counter-strategies to synthesize a set of candidate assumptions that can be used to rule out the counter-strategy. Similar to templates, patterns impose structure on the assumptions. However, our method synthesizes the patterns based on the counter-strategy and the user does not need to manipulate them. We only require the user to specify a subset of variables to be used in the search for the missing assumptions. The user can specify a subset that she thinks leads to the unrealizability. In our method, the maximum number of generated assumptions for a given counter-strategy is independent from what subset of variables is considered, whereas increasing the size of the chosen subset of variables in [7] will result in exponential growth in the number of candidates, while only a small number of them might hold over all runs of the counter-strategy (unlike our method). Moreover, we compute the weakest environment assumptions for the considered structure and given subset of variables. Our work takes an initial step toward bridging the gap between [4] and [7]. Our method synthesizes environment assumptions that are simple formulas, making them easy to understand and practical, and they also constrain the environment as weakly as possible within their structure. We refer the reader to [7] for a survey of related work.

II. Preliminaries

Linear temporal logic (LTL) is a formal specification language with two kinds of operators: logical connectives (negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$) and implication ($\rightarrow$)) and temporal modal operators (next ($\bigcirc$), always ($\Box$), eventually ($\Diamond$) and until ($\mathcal{U}$)). Given a set P of atomic propositions, an LTL formula is defined inductively as follows: 1) any atomic proposition $p \in P$ is an LTL formula. 2) if ϕ and ψ are LTL formulas, then $\neg\phi$, $\phi \vee \psi$, $\bigcirc\phi$ and $\phi\,\mathcal{U}\,\psi$ are also LTL formulas. Other operators can be defined using the following rules: $\phi \wedge \psi = \neg(\neg\phi \vee \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\Diamond\phi = \texttt{True}\,\mathcal{U}\,\phi$ and $\Box\phi = \neg\Diamond\neg\phi$. An LTL formula is interpreted over infinite words $\omega \in (2^P)^\omega$. For an LTL formula ϕ, we define its language $\mathcal{L}(\phi)$ to be the set of infinite words that satisfy ϕ, i.e., $\mathcal{L}(\phi) = \{\omega \in (2^P)^\omega \mid \omega \models \phi\}$.

A finite transition system (FTS) is a tuple $\mathcal{T} = \langle Q, Q_0, \delta\rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states and $\delta \subseteq Q \times Q$ is the transition relation. An *execution* or *run* of a FTS is an infinite sequence of states $\sigma = q_0 q_1 q_2 ...$ where $q_0 \in Q_0$ and for any $i \geq 0$, $q_i \in Q$ and $(q_i, q_{i+1}) \in \delta$. The language of a FTS $\mathcal{T}$ is defined as the set $\mathcal{L}(\mathcal{T}) = \{\omega \in Q^\omega \mid \omega \text{ is a run of } \mathcal{T}\}$, i.e., the set of (infinite) words generated by the runs of $\mathcal{T}$. We often consider a FTS as a directed graph with a natural bijection between the states and transitions of the FTS and vertices and edges of the graph, respectively. Formally for a FTS $\mathcal{T} = \langle Q, Q_0, \delta\rangle$, we define the graph $\mathcal{G}_\mathcal{T} = \langle V, E\rangle$ where each $v_i \in V$ corresponds to a unique state $q_i \in Q$, and $(v_i, v_j) \in E$ if and only if $(q_i, q_j) \in \delta$.

Let P be a set of atomic propositions, partitioned into input, I, and output, O, propositions. A *Moore transducer* is a tuple $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$, where S is the set of states, $s_0 \in S$ is the initial state, $\mathcal{I} = 2^I$ is the input alphabet, $\mathcal{O} = 2^O$ is the output alphabet, $\delta : S \times \mathcal{I} \rightarrow S$ is the transition function and $\gamma : S \rightarrow \mathcal{O}$ is the state output function. A *Mealy* transducer is similar, except that the state output function is $\gamma : S \times \mathcal{I} \rightarrow \mathcal{O}$. For an infinite word $\omega \in \mathcal{I}^\omega$, a run of M is the infinite sequence $\sigma \in S^\omega$ such that $\sigma_0 = s_0$ and for all $i \geq 0$ we have $\sigma_{i+1} = \delta(\sigma_i, \omega_i)$. The run σ on input word ω produces an infinite word $M(\omega) \in (2^P)^\omega$ such that $M(\omega)_i = \gamma(\sigma_i) \cup \omega_i$ for all $i \geq 0$. The language of M is the set $\mathcal{L}(M) = \{M(\omega) \mid \omega \in \mathcal{I}^\omega\}$ of infinite words generated by runs of M.

An LTL formula ϕ is *satisfiable* if there exists an infinite word $\omega \in (2^P)^\omega$ such that $\omega \models \phi$. A Moore (Mealy) transducer M satisfies an LTL formula ϕ, written as $M \models \phi$, if $\mathcal{L}(M) \subseteq \mathcal{L}(\phi)$. An LTL formula ϕ is *Moore (Mealy) realizable* if there exists a Moore (Mealy, respectively) transducer M such that $M \models \phi$. The *realizability problem* asks whether there exists such a transducer for a given LTL specification ϕ.

A *two-player deterministic game graph* is a tuple $\mathcal{G} = (Q, Q_0, E)$ where Q can be partitioned into two disjoint sets Q_1 and Q_2. Q_1 and Q_2 are the sets of states of player 1 and 2, respectively. Q_0 is the set of initial states. $E = Q \times Q$ is the set of directed edges. Players take turns to play the game. At each step, if the current state belongs to Q_1, player 1 chooses the next state. Otherwise player 2 makes a move. A *play* of

the game graph $\mathcal{G}$ is an infinite sequence $\sigma = q_0q_1q_2...$ of states such that $q_0 \in Q_0$, and $(q_i, q_{i+1}) \in E$ for all $i \geq 0$. We denote the set of all plays by Π. A *strategy* for player $i \in \{1, 2\}$ is a function $\alpha_i : Q^*.Q_i \to Q$ that chooses the next state given a finite sequence of states which ends at a player i state. A strategy is *memoryless* if it is a function of current state of the play, i.e., $\alpha_i : Q_i \to Q$. Given strategies α_1 and α_2 for players and a state $q \in Q$, the *outcome* is the play starting at q, and evolved according to α_1 and α_2. Formally, $outcome(q, \alpha_1, \alpha_2) = q_0q_1q_2...$ where $q_0 = q$, and for all $i \geq 0$ we have $q_{i+1} = \alpha_1(q_0q_1...q_i)$ if $q_i \in Q_1$ and $q_{i+1} = \alpha_2(q_0q_1...q_i)$ if $q_i \in Q_2$. An *objective* for a player is a set $\Phi \subseteq \Pi$ of plays. A strategy α_1 for player 1 is winning for some state q if for every strategy α_2 of player 2, we have $outcome(q, \alpha_1, \alpha_2) \in \Phi$.

Given an LTL formula ϕ over P and a partitioning of P into I and O, the *synthesis problem* is to find a Mealy transducer M with input alphabet $\mathcal{I} = 2^I$ and output alphabet $\mathcal{O} = 2^O$ that satisfies ϕ. This problem can be reduced to computing winning strategies in game graphs. A deterministic game graph G, and an objective Φ can be constructed such that ϕ is realizable if and only if the system (player 1) has a memoryless winning strategy from the initial state in G [11]. Every memoryless winning strategy of the system can be represented by a Mealy transducer that satisfies ϕ. If the specification ϕ is unrealizable, then the environment (player 2) has a winning strategy. A *counter-strategy* for the synthesis problem is a strategy for the environment that can falsify the specification, no matter how the system plays. Formally, a counter-strategy can be represented by a Moore transducer $M_c = (S', s'_0, \mathcal{I}', \mathcal{O}', \delta', \gamma')$ that satisfies $\neg\phi$, where $\mathcal{I}' = \mathcal{O}$ and $\mathcal{O}' = \mathcal{I}$ are the input and output alphabet for M_c which are generated by the system and the environment, respectively.

In this paper, we consider specifications of the form

$$\phi = \phi_e \to \phi_s, \tag{1}$$

where ϕ_α for $\alpha \in \{e, s\}$ can be written as a conjunction of the following parts:

- ϕ_i^α: A Boolean formula over I if $\alpha = e$ and over $I \cup O$ otherwise, characterizing the initial state.
- ϕ_t^α: An LTL formula of the form $\bigwedge_i \Box\psi_i$. Each subformula $\Box\psi_i$ is either characterizing an invariant, in which case ψ_i is a Boolean formula over $I \cup O$, or it is characterizing a transition relation, in which case ψ_i is a Boolean formula over expressions v and $\bigcirc v'$ where $v \in I \cup O$ and, $v' \in I$ if $\alpha = e$ and $v' \in I \cup O$ if $\alpha = s$.
- ϕ_g^α: A formula of the form $\bigwedge_i \Box\Diamond B_i$ characterizing fairness/liveness, where each B_i is a Boolean formula over $I \cup O$.

For the specifications of the form in (1), known as GR(1) formulas, Piterman et al. [10] show that the synthesis problem can be solved in polynomial time. Intuitively, in (1), ϕ_e characterizes the assumptions on the environment and ϕ_s characterizes the correct behavior (guarantees) of the system. Any correct implementation of the specification guarantees to satisfy ϕ_s, provided that the environment satisfies ϕ_e.

For a given unrealizable specification $\phi_e \to \phi_s$, we define a *refinement* $\psi = \bigwedge_i \psi_i$ as a conjunction of a collection of environment assumptions ψ_i in the GR(1) form such that $\phi_e \wedge \psi \to \phi_s$ is realizable. Intuitively it means that adding the assumptions ψ_i to the specification results in a new specification which is realizable. We say a refinement ψ is consistent with the specification $\phi_e \to \phi_s$ if $\phi_e \wedge \psi$ is satisfiable. Note that if $\phi_e \wedge \psi$ is not satisfiable, i.e., $\phi_e \wedge \psi =$ `False`, the specification $\phi_e \wedge \psi \to \phi_s$ is trivially realizable [7], but obviously ψ is not an interesting refinement.

III. Problem Statement and Overview

A. Problem Statement

Given a specification $\phi = \phi_e \to \phi_s$ in the GR(1) form which is satisfiable but unrealizable, find a refinement $\psi = \bigwedge_i \psi_i$ as a conjunction of environment assumptions ψ_i such that $\phi_e \wedge \psi$ is satisfiable and $\phi_e \wedge \psi \to \phi_s$ is realizable.

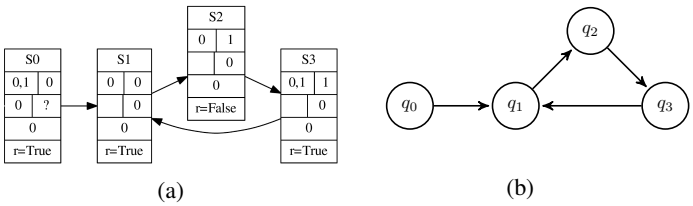
B. Overview of the Method

We now give a high-level view of our method. Specification refinements are constructed in two phases. First, given a counter-strategy's Moore machine M_c, we build an abstraction which is a FTS $\mathcal{T}_c$. The abstraction preserves the structure of the counter-strategy (its states and transitions) while removing the input and output details. The algorithm processes $\mathcal{T}_c$ and synthesizes a set of LTL formulas in special forms, called *patterns*, which hold over *all* runs of $\mathcal{T}_c$. Our algorithm then uses these patterns along with a subset of variables specified by the user to generate a set of LTL formulas which hold over *all* runs of M_c. We ask the user to specify a subset of variables which she thinks contribute to the unrealizability of the specification. This set can also be used to guide the algorithm to generate formulas over the set of variables which are underspecified. Using a smaller subset of variables leads to simpler formulas that are easier for the user to understand.

The complement of the generated formulas form the set of candidate assumptions that can be used to rule out the counter-strategy from the environment's possible behaviors. We remove the candidates which are not consistent with the specification in order to avoid a trivial solution `False`.

Any assumption from the set of generated candidates can be used to rule out the counter-strategy. Our approach does a breadth-first search over the candidates. If adding any of the candidates makes the specification realizable, the algorithm returns that candidate as a solution. Otherwise at each iteration, the process is repeated for any of the new specifications resulting from adding a candidate. The depth of the search is controlled by the user. The search continues until either a consistent refinement is found or the algorithm cannot find one within the specified depth (hence the search algorithm is sound, but not complete).

Example 1. *Consider the following example borrowed from [7] with the environment variables $I = \{r, c\}$ and system variables $O = \{g, v\}$. Here r, c, g and v stand for* request*,* clear*,* grant *and* valid *signals respectively. We start with no assumption, that is we only assume $\phi_e =$ `True`. Consider the following system guarantees: $\phi_1 = \Box(r \to \bigcirc\Diamond g)$, $\phi_2 = \Box((c \vee g) \to \bigcirc\neg g)$, $\phi_3 = \Box(c \to \neg v)$ and $\phi_4 = \Box\Diamond(g \wedge v)$.*



(a) (b)

Fig. 1: (a) A counter-strategy produced by RATSY for the specification of Example 1 with the additional assumption $\square\diamond(\neg r)$. $c =$ `True` is constant in all states. (b) The abstract finite transition system for the counter-strategy of part (a).

Let ϕ_s be the conjunction of these formulas. ϕ_1 requires that every request must be granted eventually starting from the next step by setting signal g to high. ϕ_2 says that if clear or grant signal is high, then grant must be low at the next step. ϕ_3 says if clear is high, then the valid signal must be low. Finally, ϕ_4 says that system must issue a valid grant infinitely often.

The specification $\phi_e \rightarrow \phi_s$ is unrealizable. A simple counter-strategy is for the environment to keep r and c high at all times. Then, by ϕ_3, v needs to be always low and thus ϕ_4 cannot be satisfied by any system. RATSY produces this counter-strategy which is then fed to our algorithm. An example candidate found by our algorithm to rule out this counter-strategy is the assumption $\psi = \square\diamond(\neg r)$. Adding ψ to the specification prevents the environment from always keeping r high, thus the environment cannot use the counter-strategy anymore. However, the specification $\phi_e \wedge \psi \rightarrow \phi_s$ is still unrealizable. RATSY produces the counter-strategy shown in Figure 1(a) for the new specification. The new counter-strategy keeps the c high all the times. The value of r is changed depending on the state of the counter-strategy as shown in Figure 1(a). The top block in each state of Figure 1(a) is the name of the state. RATSY produces additional information, shown in middle blocks, on how the counter-strategy enforces the system to violate the specification. We do not use this information in the current version of the algorithm.

The following formulas are examples of consistent refinements produced by our algorithm for the specification $\phi_e \rightarrow \phi_s$:

- $\psi_1 = \square(\neg r \vee \neg c) \wedge \square(r \vee \neg c)$
- $\psi_2 = \square(r \rightarrow \bigcirc\neg c) \wedge \square(\neg r \rightarrow \bigcirc\neg c)$
- $\psi_3 = \square\diamond(\neg r) \wedge \square(\neg c \vee r) \wedge \square(\neg r \rightarrow \bigcirc\neg c))$

Assumptions in both of the refinements ψ_1 and ψ_2 imply $\square(\neg c)$, that is, adding them requires the environment to keep the signal c always low. Although adding these assumptions make the specification realizable, it may not conform to the design intent. Refinement ψ_3 does not restrict c like ψ_1 and ψ_2, and only assumes that the environment sets the signal r to low infinitely often and that, when the request signal is low, the clear signal should be low at the same and the next step.

IV. Specification Refinement

Algorithm 1 finds environment assumptions that can be added to the specification to make it realizable. It gets as input the initial unrealizable specification $\phi = \phi_e \rightarrow \phi_s$, the set P of subsets of variables to be used in generated assumptions and the maximum depth α of the search. It outputs a consistent refinement ψ, if it can find one within the specified depth.

For an unrealizable specification, a counter-strategy is computed as a Moore transducer using the techniques in [5], [2]. The counter-strategy is then fed to the **GeneratePatterns** procedure which constructs a set of patterns and is detailed in Section IV-C. Procedure **GenerateCandidates**, described in Section IV-A, produces a set of candidate assumptions in the form of GR(1) formulas using patterns and the set P of variables. Algorithm 1 runs a breadth-first search to find a consistent refinement. Each node of the search tree is a generated candidate assumption, while the root of the tree corresponds to the assumption `True` (i.e., no assumption). Each path of the search tree starting from the root corresponds to a candidate refinement as conjunction of candidate assumptions of the nodes visited along the path. When a node is visited during the search, its corresponding candidate refinement is added to the specification. If the new specification is consistent and realizable, the refinement is returned by the algorithm. Otherwise, if the depth of the current node is less than the maximum specified, a set of candidate assumptions are generated based on the counter-strategy for the new specification and the search tree expands.

In Algorithm 1, the queue *CandidatesQ* keeps the candidate refinements which are found during the search. At each iteration, a candidate refinement ψ is removed from the head of the queue. The procedure **Consistent** checks if ψ is consistent with the specification ϕ. If it is, the algorithm checks the realizability of the new specification $\phi_{new} = \phi_e \wedge \psi \rightarrow \phi_s$ using the procedure **Realizable** [3], [2]. If ϕ_{new} is realizable, ψ is returned as a suggested refinement. Otherwise, if the depth of the search for reaching the candidate refinement ψ is less than α, a new set of candidate assumptions are generated using the counter-strategy computed for ϕ_{new}. Algorithm 1 keeps track of the number of counter-strategies produced along the path to reach a candidate refinement in order to compute its depth (**Depth**(ψ)). Each new candidate assumption ψ_{new} results in a new candidate refinement $\psi \wedge \psi_{new}$ which is added to the end of the queue for future processing . The algorithm terminates when either a consistent refinement ψ is found, or there is no more candidates in the queue to be processed.

A. Generating Candidates

Consider the Moore transducer $M_c = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$ of a counter-strategy, where $\mathcal{I} = 2^O$ and $\mathcal{O} = 2^I$, and O and I are the set of the system and environment variables, respectively. Given M_c, we construct a finite transition system $\mathcal{T}_c = \langle Q, \{q_0\}, \delta \rangle$ which preserves the structure of the M_c while removing all details about its input and output. More formally, for each state $s_i \in S$, $\mathcal{T}_c$ has a corresponding state $q_i \in Q$, and $q_0 \in Q$ is the state corresponding to $s_0 \in S$. There exists a transition $(q_i, q_j) \in \delta$ if and only if there exists $y \in \mathcal{I}$ such that $\delta(s_i, y) = s_j$. It is easy to see that any run of $\mathcal{T}_c$ corresponds to a run of M_c and vice versa.

By processing the abstract FTS $\mathcal{T}_c$ of the counter-strategy, we synthesize a set of patterns which are LTL formulas of the form $\diamond\square\psi_1$, $\diamond\psi_2$ and $\diamond(\psi_3 \wedge \bigcirc\psi_4)$ that hold over all runs of $\mathcal{T}_c$. Each ψ_i for $i \in \{1, 2, 3, 4\}$ is a disjunction of a subset of states of $\mathcal{T}_c$, i.e., $\psi_i = \bigvee_{q \in Q_i} q$ where $Q_i \subseteq Q$.

Algorithm 1: Specification Refinement

Input: $\phi = \phi_e \rightarrow \phi_s$, initial specification
Input: P, set of subsets of variables to be used in patterns
Input: α, maximum depth of the search
Output: ψ, additional assumptions such that $\phi_e \wedge \psi \rightarrow \phi_s$ is realizable

```
M_c := CounterStrategy(φ);
Patterns := GeneratePatterns(M_c);
CandidatesQ := GenerateCandidates(Patterns,P);
while CandidatesQ is not Empty do
    ψ := CandidatesQ.DeQueue;
    if Consistent(φ,ψ) then
        φ_new = φ_e ∧ ψ → φ_s;
        if Realizable(φ_new) then
            return ψ;
        else
            if Depth(ψ) < α then
                M_c := CounterStrategy(φ_new);
                Patterns := GeneratePatterns(M_c);
                newCandidates :=
                GenerateCandidates(Patterns,P) ;
                foreach ψ_new ∈ newCandidates do
                    CandidatesQ.EnQueue(ψ ∧ ψ_new);
return No refinement was found;
```

The complements of these formulas, $\square\diamond\neg\psi_1$ (liveness), $\square\neg\psi_2$ (safety), and $\square(\psi_3 \rightarrow \bigcirc\neg\psi_4)$ (transition), respectively, are of the desired GR(1) form and provide the structure for the candidate assumptions that can be used to rule out the counter-strategy. Note that similar to [7], we do not synthesize assumptions characterizing the initial state because they are easy to specify in practice. Besides, it is simple to discover them from the counter-strategy. Patterns are generated using simple graph search algorithms explained in Section IV-C.

Example 2. *Figure 1(b) shows the abstract FTS for the counter-strategy of Figure 1(a). For this FTS our algorithm produces the set of patterns* $\diamond\square(q_1 \vee q_2 \vee q_3)$, $\diamond q_0, \diamond q_1, \diamond q_2, \diamond q_3$, *and* $\diamond(q_0 \wedge \bigcirc q_1)$, $\diamond(q_1 \wedge \bigcirc q_2), \diamond(q_2 \wedge \bigcirc q_3), \diamond(q_3 \wedge \bigcirc q_1)$. *Any run of* $\mathcal{T}_c$ *satisfies all of the above formulas. For example* $\mathcal{T}_c \models \diamond q_i$ *for* $i \in \{0,1,2,3\}$, *meaning that any run of the* $\mathcal{T}_c$ *will eventually visit state* q_i. *The formula* $\diamond(q_1 \wedge \bigcirc q_2)$ *means that any run of* $\mathcal{T}_c$ *will eventually visit state* q_1 *and then state* q_2 *at the next step. Also any run of* $\mathcal{T}_c$ *satisfies* $\diamond\square(q_1 \vee q_2 \vee q_3)$, *meaning that any run of* $\mathcal{T}_c$ *will eventually reach and stay in the set of states* $\{q_1, q_2, q_3\}$.

As we mentioned previously, each state $q_i \in Q$ of the FTS $\mathcal{T}_c$ corresponds to a state $s_i \in S$ of the Moore transducer M_c of the counter-strategy. Also recall that each run of $\mathcal{T}_c$ corresponds to a run of M_c. M_c, at any state $s_i \in S$, outputs the propositional formula $\mathcal{V}_{s_i} = \gamma(s_i)$ which is a valuation over all environment variables. Formally, for any state $s_i \in S$ of M_c, we have $\mathcal{V}_{s_i} = \ell_1^i \wedge \ell_2^i \wedge ... \wedge \ell_n^i$ where each ℓ_j^i is a literal over the environment variable $x_j \in I$. We call $\mathcal{V}_{s_i}$ the state predicate of s_i and also q_i. We replace the states in the patterns with their corresponding state predicates to get a set of formulas which hold over all runs of the counter-strategy.

Example 3. *Consider the counter-strategy shown in Figure 1(a). The state predicates are* $\mathcal{V}_{S0} = \mathcal{V}_{S1} = \mathcal{V}_{S3} = c \wedge r$ *and* $\mathcal{V}_{S2} = c \wedge \neg r$, *where* $S0, S1, S2$ *and* $S3$ *are the states of* M_c. *Using the patterns obtained in Example 2 and replacing the states with their corresponding state predicates, we obtain LTL formulas which hold over all runs of* M_c. *For example, the pattern* $\diamond\square(q_1 \vee q_2 \vee q_3)$ *gives us the formula* $\diamond\square((c \wedge r) \vee (c \wedge \neg r)) = \diamond\square c$. *Replacing* q_2 *with* $\mathcal{V}_{S2}$ *in the pattern* $\diamond q_2$ *leads to* $\diamond(c \wedge \neg r)$. *Similarly, the pattern* $\diamond(q_1 \wedge \bigcirc q_2)$ *gives* $\diamond((c \wedge r) \wedge \bigcirc(c \wedge \neg r))$.

The structure of the state predicates and patterns is such that any subset of the environment variables can be used along with the patterns to generate candidates and the resulting formulas still hold over all runs of the counter-strategy. Algorithm 1 gets the set $P = \{P_1, P_2, P_3, P_4\}$ as input, where each P_i is a subset of environment variables that should be used in the corresponding ψ_i for generating the candidate assumptions from the patterns of the form $\diamond\square\psi_1$, $\diamond\psi_2$ and $\diamond(\psi_3 \wedge \bigcirc\psi_4)$.

Example 4. *Assume that the designer specifies* $P_1 = \{r\}$, $P_2 = \{c\}$, $P_3 = \{r, c\}$ *and* $P_4 = \{c\}$. *Then the pattern* $\diamond\square(q_1 \vee q_2 \vee q_3)$ *results in* $\diamond\square(r \vee \neg r \vee r) = \diamond\square$`True`. *From* $\diamond q_2$ *we obtain* $\diamond c$, *and* $\diamond(q_1 \wedge \bigcirc q_2)$ *leads to* $\diamond((c \wedge r) \wedge \bigcirc c)$. *Note that using a smaller subset of variables leads to simpler formulas (and sometimes trivial as in* $\diamond\square($`True`$)$*). However, this simplicity may result in assumptions which put more constraints on the environment as we will show later.*

The complement of the generated formulas form the set of candidate assumptions that can be used to rule out the counter-strategy. For instance, formulas $\square\diamond(\neg r \wedge r) = \square\diamond($`False`$)$, $\square(\neg c)$, $\square((c \wedge r) \rightarrow \bigcirc(\neg c))$ and $\square((c \wedge \neg r) \rightarrow \bigcirc(\neg c))$ are the candidate assumptions computed based on the user input in Example 4. Note that there might be repetitive formulas among the generated candidates. We remove the repeated formulas in order to prevent the process from checking the same assumption repeatedly. We also use some techniques to simplify the synthesized assumptions (see [1]).

B. Removing the Restrictive Formulas

Given two non-equivalent formulas ϕ_1 and ϕ_2 we say ϕ_1 is *stronger* than ϕ_2 if $\phi_1 \rightarrow \phi_2$ holds. Assume ψ_1 and ψ_2 are two formulas that hold over all runs of the counter-strategy computed for the specification $\phi_e \rightarrow \phi_s$, and that $\psi_1 \rightarrow \psi_2$. Note that $\neg\psi_2 \rightarrow \neg\psi_1$ also holds, that is $\neg\psi_1$ is a *weaker* assumption compared to $\neg\psi_2$. Adding either $\neg\psi_1$ or $\neg\psi_2$ to the environment assumptions ϕ_e rules out the counter-strategy. However, adding the stronger assumption $\neg\psi_2$ restricts the environment more than adding $\neg\psi_1$. That is, $\phi_e \wedge \neg\psi_2$ puts more constraints on the environment compared to $\phi_e \wedge \neg\psi_1$.

As an example, consider the counter-strategy M_c shown in Figure 1(a). Both $\psi_1 = \diamond(c \wedge \neg r)$ and $\psi_2 = \diamond(c)$ hold over all runs of M_c. Moreover, $\psi_1 \rightarrow \psi_2$. Consider the corresponding assumptions $\neg\psi_1 = \square(\neg c \vee r)$ and $\neg\psi_2 = \square(\neg c)$. Adding $\neg\psi_2$ restricts the environment more than adding $\neg\psi_1$. $\neg\psi_2$ requires the environment to keep the signal c always low, whereas in case of $\neg\psi_1$, the environment is free to assign additional values to its variables. It only prevents the environment from setting c to high and r to low at the same time.

We construct patterns which are strongest formulas of their specified form that hold over all runs of the counter-

strategy. Therefore, the generated candidate assumptions are the weakest formulas that can be constructed for the given structure and the user specified subset of variables.

C. Synthesizing Patterns

In this section we show how certain types of patterns can be synthesized using the abstract FTS $\mathcal{T}_c$ of the counter-strategy. A pattern $\mathcal{P}$, is an LTL formula $\phi_{\mathcal{P}}$ which holds over all runs of the FTS $\mathcal{T}_c$, i.e., $\mathcal{T}_c \models \phi_{\mathcal{P}}$. We are interested in patterns of the form $\diamond\square\psi$, $\diamond\psi$ and $\diamond(\psi_1 \wedge \bigcirc\psi_2)$. The complements of these patterns are of the GR(1) form and, after replacing states with their corresponding state predicates, will yield to candidate assumptions for removing the counter-strategy.

1) Patterns of the Form $\diamond\psi$: For a FTS $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$, we define a *configuration* $C \subseteq Q$ as a subset of states of $\mathcal{T}_c$. We say a configuration C is an *eventually configuration* if for any run σ of $\mathcal{T}_c$ there exists a state $q \in C$ and a time step $i \geq 0$ such that $\sigma_i = q$. That is, any run of $\mathcal{T}_c$ eventually visits a state from the configuration C. It follows that if C is an eventually configuration for $\mathcal{T}_c$, then $\mathcal{T}_c \models \diamond \bigvee_{q\in C} q$. We say an eventually configuration C is *minimal* if there exists no $C' \subset C$ such that C' is an eventually configuration. Note that removing any state $q \in C$ from a minimal eventually configuration leads to a configuration which is not an eventually configuration.

Algorithm 2 constructs eventually patterns which correspond to the minimal eventually configurations of $\mathcal{T}_c$ with size less than or equal to β. The larger configurations lead to larger formulas which are hard for the user to parse. The user can specify the value of β. Heuristics can also be used to automatically set β based on the properties of $\mathcal{T}_c$, e.g. the maximum outdegree of the vertices in the corresponding directed graph $\mathcal{G}_{\mathcal{T}_c}$, where the outdegree of a vertex is the number of its outgoing edges. In Algorithm 2, the set $\diamond$Configurations keeps the minimal eventually configurations discovered so far. Algorithm 2 initializes the sets Patterns and $\diamond$Configurations to $\{\diamond q_0\}$ and $\{q_0\}$, respectively. Note that $\diamond q_0$ holds over all runs of $\mathcal{T}_c$. The algorithm then checks each possible configuration $Q' \subseteq Q - \{q_0\}$ with size less than or equal to β in a non-decreasing order of $|Q'|$ to find minimal eventually configurations. Without loss of generality we assume that all states in $\mathcal{T}_c$ have outgoing edges[1]. At each iteration, a configuration Q' is chosen. Algorithm 2 checks if there is a minimal eventually configuration Q'' which is already discovered and $Q'' \subset Q'$. If such Q'' exists, Q' is not minimal. Otherwise, the algorithm checks if it is an eventually configuration by first removing all the states in Q' and their corresponding incoming and outgoing transitions from $\mathcal{T}_c$ to obtain another FTS $\mathcal{T}_c'$. Now, if there is an infinite run from q_0 in $\mathcal{T}_c'$, then there is a run in $\mathcal{T}_c$ that does not visit any state in Q'. Otherwise, Q' is a minimal eventually configuration and is added to $\diamond$Configurations. The corresponding formula $\psi = \diamond \bigvee_{q\in Q'} q$ is also added to the set of eventually patterns. Note that checking if there exists an infinite run in $\mathcal{T}_c'$ can be done by considering $\mathcal{T}_c'$ as a graph and checking if there is a reachable cycle from q_0, which can be done in linear time in number of states and transitions of $\mathcal{T}_c$. Therefore, the algorithm is of complexity $O(|Q|^{\beta}(|Q| + |\delta|))$.

[1]A transition from any state with no outgoing transition can be added to a dummy state with a self loop. Patterns which include the dummy state will be removed.

Algorithm 2: Generating $\diamond\psi$ patterns

Input: Finite state transition system $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$
Input: β, maximum number of states in generated patterns
Output: a set of patterns of the form $\diamond\psi$ where $\mathcal{T}_c \models \diamond\psi$

```
Patterns := {◇q_0};
◇Configurations := {q_0};
foreach Q' ⊆ Q − {q_0} with non-decreasing order of |Q'| where |Q'| ≤ β do
    if ∄Q'' ∈ ◇Configurations s.t. Q'' ⊆ Q' then
        Let T_c' = ⟨Q − Q', {q_0}, δ'⟩ where
        δ' = {(q, q') ∈ δ | q ∉ Q' ∧ q' ∉ Q'};
        if there is no infinite run from q_0 in T_c' then
            Add Q' to ◇Configurations;
            Let ψ = ◇ ⋁_{q_i∈Q'} q_i;
            Add ψ to Patterns;
return Patterns;
```

Example 5. *Consider the FTS shown in Figure 2. Algorithm 2 starts at initial configuration $\{q_0\}$ and generates the formula $\diamond q_0$. None of $\{q_1\}$, $\{q_2\}$ or $\{q_3\}$ is an eventually configuration. For example for configuration $\{q_1\}$, there exists the run $\sigma = q_0, (q_3)^{\omega}$ which never visits q_1. Configurations $\{q_1, q_3\}$ and $\{q_2, q_3\}$ are minimal eventually configurations. For example removing $\{q_1, q_3\}$ will lead to a FTS with no infinite run (no cycle is reachable from q_0 in the corresponding graph). It is easy to see that configuration $\{q_1, q_2\}$ is not an eventually configuration. Configuration $\{q_1, q_2, q_3\}$ is not minimal, although it is an eventually configuration. Thus Algorithm 2 returns the set of patterns $\{\diamond q_0, \diamond(q_1 \vee q_3), \diamond(q_2 \vee q_3)\}$.*

2) Patterns of the Form $\diamond\square\psi$: To compute formulas of the form $\diamond\square\psi$ which hold over all runs of the FTS $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$ of the counter-strategy, we view $\mathcal{T}_c$ as a graph and separate its states into two groups: $Q^{cycle} \subseteq Q$, the set of states that are part of a cycle in $\mathcal{T}_c$ (including the cycle from one node to itself), and $Q' = Q - Q^{cycle}$. Without loss of generality we assume that any state $q \in Q$ is reachable from q_0. Therefore, any state $q \in Q^{cycle}$ belongs to a reachable strongly connected component C of $\mathcal{T}_c$. Also for any strongly connected component C of $\mathcal{T}_c$, there exists a run σ of $\mathcal{T}_c$ which reaches states in C and keeps cycling there forever. Hence, the formula $\psi_1 = \diamond\square \bigvee_{q\in C} q$ holds over the run σ. Indeed ψ_1 is the minimal formula of disjunctive form which holds over all runs that can reach the strongly connected component C. That is, by removing any of the states from ψ_1, one can find a run σ' which can reach the strongly connected component C and visit the removed state, falsifying the resulted formula. Therefore, eventually for any execution of $\mathcal{T}_c$, the state of the system will always be in one of the states $q \in Q^{cycle}$. Thus the formula $\psi = \diamond\square \bigvee_{q\in Q^{cycle}} q$ is the minimal formula of the form eventually always which holds over all runs of $\mathcal{T}_c$.

To partition the states of the $\mathcal{T}_c$ into Q^{cycle} and Q' we use Tarjan's algorithm for computing strongly connected components of the graph. Thus the algorithm is of linear time

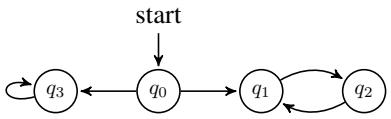


Fig. 2: A non-deterministic finite state transition system $\mathcal{T}_c$

complexity in number of states and transitions of $\mathcal{T}_c$.

Example 6. *Consider the non-deterministic FTS shown in Figure 2. It has three strongly connected components:* $\{q_0\}$, $\{q_1, q_2\}$ *and* $\{q_3\}$. *Only the latter two components include a cycle inside them, that is* $Q^{cycle} = \{q_1, q_2, q_3\}$. *Thus, the pattern* $\psi = \diamond\Box(q_1 \vee q_2 \vee q_3)$ *is generated. Note that the possible runs of the system are* $\sigma_1 = q_0, (q_1, q_2)^\omega$ *and* $\sigma_2 = q_0, (q_3)^\omega$. *The generated pattern* ψ *holds over both of these runs. Observe that removing any of the states in* ψ *will result in a formula which is not satisfied by* $\mathcal{T}_c$ *any more.*

3) Patterns of the Form $\diamond(\psi_1 \wedge \bigcirc\psi_2)$*:* To generate candidates of the form $\diamond(\psi_1 \wedge \bigcirc\psi_2)$, first note that $\diamond(\psi_1 \wedge \bigcirc\psi_2)$ holds only if $\diamond\psi_1$ holds. Therefore, a set of eventually patterns $\diamond\psi_1$ is first computed using Algorithm 2. Then for each formula $\diamond\psi_1$, the pattern $\diamond(\psi_1 \wedge \bigcirc \bigvee_{q \in Next(\psi_1)} q)$ is generated, where $Next(\psi_1)$ is the set of states that can be reached in one step from the configuration specified by ψ_1. Formally, $Next(\psi_1) = \{q_i \in Q \mid \exists q_j \in \mathcal{C} \text{ s.t. } (q_j, q_i) \in \delta\}$ and $\mathcal{C}$ is the configuration represented by $\psi_1 = \bigvee_{q \in \mathcal{C}} q$. The most expensive part of this procedure is computing the eventually patterns, therefore its complexity is the same as Algorithm 2. Algorithms for computing $\diamond\Box\psi$ and $\diamond(\psi_1 \wedge \bigcirc\psi_2)$ patterns can be found in the technical report [1].

Example 7. *Consider the FTS shown in Figure 2. Given the set of eventually formulas produced in Example 5, patterns* $\diamond(q_0 \wedge \bigcirc(q_1 \vee q_3))$, $\diamond((q_1 \vee q_3) \wedge \bigcirc(q_2 \vee q_3))$ *and* $\diamond((q_2 \vee q_3) \wedge \bigcirc(q_1 \vee q_3))$ *are generated.*

The procedures described for producing patterns, lead to assumptions which only include environment variables, and are enough for resolving unrealizability in our case studies. However, in general, GR(1) assumptions can also include the system variables. The procedures can be easily extended to the general case (see [1]).

The following theorem states that the procedures described in this section, generate the strongest patterns of the specified forms. Its proof can be found in [1]. Removing the weaker patterns leads to shorter formulas which are easier for the user to understand. It also decreases the number of generated candidates at each step. More importantly, it leads to weaker assumptions on the environment that can be used to rule out the counter-strategy. If the restriction imposed by any of these candidates is not enough to make the specification realizable, the method analyzes the counter-strategy computed for the new specification to find assumptions that can restrict the environment more. This way the counter-strategies guide the method to synthesize assumptions that can be used to achieve realizability.

Theorem 1. *For any formula of the form* $\diamond\psi, \diamond\Box\psi$, *or* $\diamond(\psi_1 \wedge \bigcirc\psi_2)$ *which hold over all runs of a given FTS* $\mathcal{T}_c$, *there is an equivalent or stronger formula of the same form synthesized by the algorithms described in Section IV-C.*

V. Case studies

We now present two case studies. We use RATSY to generate counter-strategies and Cadence SMV model checker [8] to check the consistency of the generated candidates. In our experiments, we set α in Algorithm 1 to two, and β in Algorithm 2 to the maximum outdegree of the vertices of the counter-strategy's abstract directed graph. We slightly change Algorithm 1 to find all possible refinements within the specified depth.

A. Lift Controller

We borrow the lift controller example from [3]. Consider a lift controller serving three floors. Assume that the lift has three buttons, denoted by the Boolean variables b_1, b_2 and b_3, which are controlled by the environment. The location of the lift is represented using Boolean variables f_1, f_2 and f_3 controlled by the system. The lift may be requested on each floor by pressing the corresponding button. We assume that (1) once a request is made, it cannot be withdrawn, (2) once the request is fulfilled it is removed, and (3) initially there are no requests. Formally, the specification of the environment is $\phi_e = \phi^e_{init} \wedge \phi^e_{1_1} \wedge \phi^e_{1_2} \wedge \phi^e_{1_3} \wedge \phi^e_{2_1} \wedge \phi^e_{2_2} \wedge \phi^e_{2_3}$, where $\phi^e_{init} = (\neg b_1 \wedge \neg b_2 \wedge \neg b_3)$, $\phi^e_{1_i} = \Box(b_i \wedge f_i \rightarrow \bigcirc\neg b_i)$, and $\phi^e_{2_i} = \Box(b_i \wedge \neg f_i \rightarrow \bigcirc b_i)$ for $1 \leq i \leq 3$.

The lift initially starts on the first floor. We expect the lift to be only on one of the floors at each step. It can move at most one floor at each time step. We want the system to eventually fulfill all the requests. Formally the specification of the system is given as $\phi_s = \phi^s_{init} \wedge \phi^s_1 \bigwedge_i \phi^s_{2,i} \wedge \phi^s_3 \bigwedge_j \phi^s_{4,j} \wedge \phi^s_5$, where

- $\phi^s_{init} = f_1 \wedge \neg f_2 \wedge \neg f_3$,
- $\phi^s_1 = \Box(\neg(f_1 \wedge f_2) \wedge \neg(f_2 \wedge f_3) \wedge \neg(f_1 \wedge f_3))$,
- $\phi^s_{2,i} = \Box(f_i \rightarrow \bigcirc(f_{i-1} \vee f_i \vee f_{i+1}))$,
- $\phi^s_3 = \Box((f_1 \wedge \bigcirc f_2) \vee (f_2 \wedge \bigcirc f_3) \rightarrow (b_1 \vee b_2 \vee b_3))$, and
- $\phi^s_{4,j} = \Box\diamond(b_j \rightarrow f_j)$.

The requirement ϕ^s_3 says that the lift moves up one floor only if some button is pressed. The specification $\phi = \phi_e \rightarrow \phi_s$ is realizable. Now assume that the designer wants to ensure that all floors are infinitely often visited; thus she adds the guarantees $\bigwedge_j \phi^s_{5,j}$ where $\phi^s_{5,j} = \Box\diamond(f_j)$ to the set of system requirements. The specification $\phi' = \phi_e \rightarrow \phi_s \bigwedge_j \phi^s_{5,j}$ is not realizable. A counter-strategy for the environment is to always keep all b_i's low. We run our algorithms with the set of all the environment variables $\{b_1, b_2, b_3\}$ for all assumption forms. The algorithm generates the refinements $\psi_1 = \Box\diamond(b_1 \vee b_2 \vee b_3)$ and $\psi_2 = \Box((\neg b_1 \wedge \neg b_2 \wedge \neg b_3) \rightarrow \bigcirc(b_1 \vee b_2 \vee b_3))$. Refinement ψ_1 requires that the environment infinitely often presses a button. Refinement ψ_2 is another suggestion which requires the environment to make a request after any inactive turn. Refinement ψ_1 seems to be more reasonable and the user can add it to the specification to make it realizable.

Only one counter-strategy is processed during the search for finding refinements and three candidate assumptions are generated overall, where one of the candidates is inconsistent with ϕ' and the two others are refinements ψ_1 and ψ_2. Thus, the search terminates after checking the generated assumptions at

first level. Only 0.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategy. Note that to generate ψ_1 using the template-based method in [7], the user needs to specify a template with three variables which leads to $2^3 = 8$ candidate assumptions, although only one of them is satisfied by the counter-strategy.

B. AMBA AHB

ARM's Advanced Microcontroller Bus Architecture (AMBA) defines the Advanced High-Performance Bus (AHB) which is an on-chip communication protocol. Up to 16 *masters* and 16 *slaves* can be connected to the bus. The masters start the communication (read or write) with a slave and the slave responds to the request. Multiple masters can request the bus at the same time, but the bus can only be accessed by one master at a time. A bus access can be a single *transfer* or a *burst*, which consists of multiple number of transfers. A bus access can be locked, which means it cannot be interrupted. Access to the bus is controlled by the *arbiter*. More details of the protocol can be found in [3]. We use the specification given by one of RATSY's example files (amba02.rat). There are four environment signals:

- HBUSREQ[i]: Master i requests access to the bus.
- HLOCK[i]: Master i requests a locked access to the bus. This signal is raised in combination with HBUSREQ[i].
- HBURST[$1:0$]: Type of transfer. Can be SINGLE (a single transfer), BURST4 (a four-transfer), or INCR (unspecified length burst).
- HREADY: Raised if the slave has finished processing the data. The bus owner can change and transfers can start only when HREADY is high.

The first three signals are controlled by the masters and the last one is controlled by the slaves. The specification of amba02.rat consists of one master and two slaves. For our experiment, we remove the fairness assumption $\square\diamond$HREADY from the specification. The new specification is unrealizable. We run our algorithm with the sets of variables $\{$HREADY$\}$, $\{$HREADY, HBUSREQ[0], HBUSREQ[1], HLOCK[0], HLOCK[1]$\}$, $\{$HREADY$\}$ and $\{$HBUSREQ[0], HBUSREQ[1]$\}$ to be used in liveness, safety, left and right hand side of transition assumptions, respectively. Some of the refinements generated by our method are: $\psi_1 = \square\diamond$HREADY, $\psi_2 = \square($HREADY $\vee \neg$HBUSREQ[0] $\vee \neg$HLOCK[0] $\vee \neg$HBUSREQ[1] $\vee \neg$HLOCK[1]$) \wedge \square\diamond$HREADY, and $\psi_3 = \square($HREADY $\rightarrow \bigcirc\neg$HBUSREQ[0]$) \wedge \square(\neg$HREADY $\rightarrow \bigcirc\neg$HBUSREQ[0]$)$. Note that although ψ_2 is a consistent refinement, it includes ψ_1 as a subformula and it is more restrictive. The refinement ψ_3 implies that HBUSREQ[0] must always be low from the second step on. Among these suggested refinements, ψ_1 appears to be the best option. Our method only processes one counter-strategy with five states and generates five candidate assumptions to find the first refinement ψ_1. To find all refinements within the depth two, overall five counter-strategies are processed by our method during the search, where the largest counter-strategy had 25 states. The number of assumptions generated for each counter-strategy during the search is less than nine. 28.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategies.

VI. Conclusion and Future Work

We presented a counter-strategy guided approach for adding environment assumptions to an unrealizable specifications in order to achieve realizability. We gave algorithms for synthesizing weakest assumptions of certain forms (based on "patterns") that can be used to rule out the counter-strategy.

We chose to apply explicit-state graph search algorithms on the counter-strategy because the available tools for solving games output the counter-strategy as a graph in an explicit form. Symbolic analysis of the counter-strategy may be desirable for scalability, but the key challenge for this is to develop algorithms for solving games that can produce counter-examples in compact symbolic form. Synthesizing symbolic patterns is one of the future directions.

Counter-strategies provide useful information for explaining reasons for unrealizability. However, there can be multiple ways to rule out a counter-strategy. We plan to investigate how the multiplicity of the candidates generated by our method can be used to synthesize better assumptions. Furthermore, our method asks the user for subsets of variables to be used in generating candidates. The choice of the subsets can significantly impact how fast the algorithm can find a refinement. Automatically finding good subsets of variables that contribute to the unrealizability problem is another future direction. Synthesizing environment assumptions for more general settings, and using the method for synthesizing interfaces between components in context of compositional synthesis are subject to our current work.

References

[1] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of gr(1) temporal logic specifications. Technical report. arXiv:1308.4113 [cs.LO].

[2] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy–a new requirements analysis tool with synthesis. In *CAV 2010*, pages 425–429. Springer, 2010.

[3] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[4] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *CONCUR 2008*, pages 147–161. Springer, 2008.

[5] R. Konighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD 2009*, pages 152–159, 2009.

[6] O. Kupferman, N. Piterman, and M. Vardi. Safraless compositional synthesis. In *CAV 2006*, pages 31–44. Springer, 2006.

[7] W. Li, L. Dworkin, and S. Seshia. Mining assumptions for synthesis. In *MEMOCODE 2011*, pages 43–50. IEEE, 2011.

[8] K. McMillan. Cadence SMV. http://www.kenmcmil.com/smv.html.

[9] N. Ozay, U. Topcu, and R. Murray. Distributed power allocation for vehicle management systems. In *CDC-ECC 2011*, pages 4841–4848. IEEE, 2011.

[10] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In *VMCAI 2006*, pages 364–380. Springer, 2006.

[11] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL 1989*, pages 179–190. ACM, 1989.

[12] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.

Efficient Handling of Obligation Constraints in Synthesis from Omega-Regular Specifications

Saqib Sohail and Fabio Somenzi
University of Colorado at Boulder

***Abstract*—A finite state reactive system (for instance a hardware controller) can be specified through a set of ω-regular properties, most of which are often safety properties. In the game-based approach to synthesis, the specification is converted to a game between the system and the environment. A deterministic implementation is obtained from the game graph and a system's winning strategy. However, there are obstacles to extract an efficient implementation from the game in hardware. On the one hand, a large space must be explored to find a strategy that has a concise representation. On the other hand, the transition structure inherited from the game graph may correspond to a state encoding that is far from optimal.**

In the approach presented in this paper, the game is formulated as a sequence of Boolean equations. That leads to significant improvements in the quality of the implementation compared to existing automata-based techniques. It is also shown discussed to extend this approach to the synthesis from obligation properties.

I. Introduction

Synthesizing reactive systems from ω-regular specifications [20] allows designers to focus on intended behavior rather than implementation details. Acceptance of automated techniques, however, is in proportion to their ability to deliver designs that meet cost and performance targets and are comparable to those produced by humans.

We present techniques that increase the performance of synthesis algorithms and lead to more compact implementations. While the algorithms are general, our implementation is geared towards the synthesis of hardware controllers. We assume that the specification of the reactive system to be synthesized is given by a list of ω-regular properties (system guarantees) that must hold when the environment satisfies another list of ω-regular properties (environment assumption). Each property is first translated to a deterministic parity automaton [18]. In special cases—like specifications in general reactive(1) form [19]—our algorithms take full advantage of this restricted form of specification. However, we do not impose restrictions on the input specification beyond ω-regularity.

Our approach extends the one of [21] in several ways. We synthesize safety properties by first reducing them to relation constraints and then manipulating the constraints in a fully symbolic form. Unlike previous techniques, this approach does not tend to embed the structures of the property automata in the implementation and produces designs with fewer state bits, better state encoding, and simpler combinational logic. It can also be considerably faster than techniques that work on explicit representations of the automata, especially when many safety properties are combined. The result of the process is a parameterized system suitable for incremental synthesis. This is useful when the specification has more than just safety or obligation properties.

We extend the symbolic approach based on transition constraint to the synthesis from obligation properties [14], which include implications between safety properties. For such implications, in particular, we show how to reduce the synthesis game to two safety games based on transition constraints.

The typical approach of automatic synthesis from the specification derives deterministic automata for each safety property; these automata operate in parallel to constrain the transitions of the system. The transition function of the composition of these automata is then inherited by the implementation. In this paper, we propose a novel way to extract the transition function of the implementation. When all the safety properties describe a language that can be generated by a relation so that the problem of sequential synthesis is converted to a problem of combination synthesis. Otherwise, we add just enough memory so that the conversion is correct. We then solve the problem of combinational synthesis by solving Boolean equations. The general solutions capture all the possible ways in which the system can satisfy the safety properties in the specification. The parameterized representation of the general solutions allows us to take advantage of the incremental synthesis framework of [21]. An additional advantage of our approach is that it is symbolic and thus adept at manipulating a large set of safety properties.

The size of the synthesized implementation depends on the transition function of the game and on the system's winning strategy. The authors in [1] provide a heuristic to select a winning strategy that attempts to minimize the amount of combinational logic in the implementation. On the other hand, the authors in [6] provide a heuristic to select a winning strategy that attempts to minimize the amount of sequential logic. However, improving the symbolic representation of the game obtained from the specification is not the focus of these works. The importance of efficient game representation has also been observed in [13]. The author remarks that current techniques are unable to extract an efficient transition structure of the implementation and proposes a tree-based approach to reduce the dependency of the implementation on the syntax of the specification. No experiments are reported.

The paper is organized as follows: Sec. II covers background and introduces notation. Sec. III, IV and V show that safety properties can be generated by a relation. Sec. VI and VII

This work was supported in part by SRC contract 2012-TJ-2271.

ISBN 978-0-9835678-3-7/13.

discuss the synthesis from safety properties from such a relation. Sec. VIII extends the relation-based approach to obligation properties. Experimental results are described in Sec. IX and conclusions are drawn in Sec. X.

II. Preliminaries

A. Linear-Time Properties

A finite automaton on ω-words $\langle \Sigma, Q, q_{\text{in}}, \delta, \alpha \rangle$ (an ω-automaton) is defined by a finite alphabet Σ, a finite set of states Q, an initial state $q_{\text{in}} \in Q$, a transition function $\delta : Q \times \Sigma \to 2^Q$ that maps a state and an input letter to a set of possible successors, and an acceptance condition α that describes a subset of Q^ω, that is, a set of infinite sequences of states. A *deterministic* automaton is such that $\delta(q, \sigma)$ is empty or a singleton for all states $q \in Q$ and all letters $\sigma \in \Sigma$. (In the case $\delta(q, \sigma)$ is a singleton, we write $\delta(q, \sigma) = q'$ for $\delta(q, \sigma) = \{q'\}$.) A run of automaton A on ω-word $w = w_0 w_1 \ldots$ is a sequence $q_0, q_1, \ldots$ such that $q_0 = q_{\text{in}}$, and for $i \geq 0$, $q_{i+1} \in \delta(q_i, w_i)$. A run is accepting iff (if and only if) it belongs to the set described by α, and a word is accepted iff it has an accepting run in A. The subset of Σ^ω accepted by A is the language of A, written $L(A)$.

Several types of acceptance conditions α are in use. We are concerned with parity conditions [16], [8], which concern the set of states $\mathsf{inf}(\rho)$ that occur infinitely often in a run ρ. A parity condition assigns a *priority* to each state: a condition of index k is a function $\pi : Q \to \{i \mid 0 \leq i \leq k\}$. A run ρ is accepting iff $\max\{\pi(q) \mid q \in \mathsf{inf}(\rho)\}$ is odd; that is, iff the highest recurring priority is odd [5], [9]. A deterministic parity word automaton (DPW) is a deterministic ω-automaton equipped with a parity condition; it is of minimum index if there is no other DPW for the same language with an acceptance condition of lower index. In the sequel, parity automata are deterministic and of minimum index [5]. A conjunctive parity condition is a set of parity conditions; and a run is accepting iff it is accepting by every parity condition. A disjunctive parity condition is a set of parity conditions; and a run is accepting iff it is accepting by some parity condition. Automata are also assumed to be *reduced*: all states are reachable from the initial state and the language accepted from them is nonempty.

We fix a finite set of atomic propositions X and consider the alphabet $\Sigma = 2^X$. An ω-regular linear-time property is a subset of Σ^ω that is accepted by a DPW. A linear-time *safety* property is a closed set of the product topology of Σ^ω. Safety properties are accepted by DPWs of index 2 such that there is no path from priority 0 states to priority 1 states. Non-safety properties are *progress* properties [15]. A safety automaton is *irredundant* if no two states accept the same language.

Linear-time temporal logic (LTL) is a specification mechanism that can express a subset of the ω-regular properties. Formulae of LTL are built from the atomic propositions in X by applying Boolean connectives and temporal operators U (until), R (releases), and X (next). Convenient abbreviations include G (globally), F (eventually), and W (weak until). An LTL formula is in negation normal form if negation is restricted to atomic propositions. The language described by the LTL formula ϕ is denoted by $L(\phi)$.

B. Realizability, Synthesis, and Games

An ω-regular property W is *satisfiable* if it has a model. For a given partition (X_e, X_s) of the atomic propositions, W is *realizable* if there exists a winning strategy for the player controlling X_s (the system) in the following game against the player controlling X_e (the environment): at each turn the environment and the system choose subsets of the propositions they control, jointly selecting an element of Σ. The elements chosen at successive turns form an infinite sequence $\rho \in \Sigma^\omega$. If ρ is in W, then the system wins; otherwise the environment wins. If the system has a winning strategy, then W is realizable, in which case a program or circuit satisfying W can be extracted from the strategy.

A full specification of the game requires detailing what each player knows of the opponent's choices when making its own choices. Variants of realizability result from different assumptions: If the system is fully apprised of the environment's choice for the same turn, *Mealy* realizability is obtained. In the opposite case, *Moore* realizability follows. We adopt a formulation that encompasses both Mealy and Moore realizability as special cases.

An ω-regular property ϕ over X specifying a reactive system is accepted by a DPW $A_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$. To check the realizability of ϕ, A_ϕ is interpreted as an input-based parity game $G_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$, where $X = X_{ed} \cup X_s \cup X_{ep}$, and Σ is the Cartesian product of a disclosed environment alphabet $\Sigma_{ed} = 2^{X_{ed}}$, a system alphabet $\Sigma_s = 2^{X_s}$, and a private environment alphabet $\Sigma_{ep} = 2^{X_{ep}}$. When the token is in state $q \in Q$, the environment chooses a letter σ_{ed} and discloses it to the system; then the system chooses a letter σ_s and discloses it to the environment; finally the environment selects a letter σ_{ep} and the token moves to state $q' = \delta(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))$. If the system has a strategy $\tau_s : S \times Q \times \Sigma_{ed} \to \Sigma_s \times S$ to win this game from q_{in} then ϕ is realizable[1]. The set S is the system's memory.

III. $\mathcal{R}$-Generability

This section is concerned with the safety languages that can be generated by a relation on the alphabet Σ. It characterizes the $\mathcal{R}$-generable languages in terms of the automata that accept them and establishes the correspondence between the automata-based view and the linguistic view of [7]. The automata-based approach provides efficient membership tests that are used in subsequent sections to devise an efficient synthesis procedure for properties that are $\mathcal{R}$-generable.

Definition 1. A set of infinite words $W \subseteq \Sigma^\omega$ is *$\mathcal{R}$-generable* if there exists a binary relation R on Σ such that a sequence $w_0 w_1 w_2 \ldots$ is in W iff $\forall i \geq 0$, (w_i, w_{i+1}) is in R.

The subset of Σ^ω generated by $R \subseteq \Sigma \times \Sigma$ is denoted by $L(R)$. It has been shown in [7] that a set of infinite words $W \subseteq \Sigma^\omega$ is $\mathcal{R}$-generable iff it is suffix-closed, fusion-closed, and limit-closed[2]. These concepts are defined as follows:

[1] If $|\Sigma_{ed}| = 1$ then system's winning strategy has a *Moore* implementation.

[2] In the context of synthesis, it is convenient to drop the requirement that the relation be total. As part of the realizability check of a specification, a subset of the alphabet is computed over which the relation is indeed total.

Definition 2. The language $W \subseteq \Sigma^\omega$ is *suffix-closed* if for every word $w_0w_1w_2\ldots \in W$ then the suffix $w_1w_2\ldots$ is in W. The language $W \subseteq \Sigma^\omega$ is *fusion-closed* if the words xvy and avb are in W, then $xvb \in W$ (and $avy \in W$). The language $W \subseteq \Sigma^\omega$ is *limit-closed* if whenever the words w_0a, w_0w_1b, $w_0w_1w_2c$, ... belong to W, then the limit of the prefixes $w_0, w_0w_1, w_0w_1w_2, \ldots$, which is the infinite word $w_0w_1w_2\ldots$ is also in W.

Limit-closed ω-regular languages are accepted by safety automata [11]. The structure of the ω-automata that recognize suffix-closed and fusion-closed languages are now examined. For lack of space, most proofs are omitted, except those that provide constructions used in the algorithms.

Definition 3. An automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi\rangle$ is *initially free* iff $\forall\sigma \in \Sigma\,.\,\delta(q_{\text{in}}, \sigma) = \{q' \mid \exists q \in Q\,.\,\delta(q,\sigma) = q'\}$.

Lemma 1. *An ω-regular language $W \subseteq \Sigma^\omega$ is suffix-closed iff it is accepted by an initially-free automaton over Σ.*

To check whether an ω-automaton A accepts a suffix-closed language, one constructs an initially-free automaton A' as described in the proof of Theorem 1 below. If $L(A') \subseteq L(A)$ then $L(A)$ is suffix-closed. When A is a deterministic safety automaton, if its initial state simulates every other state then $L(A)$ is suffix-closed.

Definition 4. An ω-automaton A is *1-definite* if the current state of A is determined by the most recent letter read.

The following result is a special case of the test for definiteness [17], [10]:

Lemma 2. *An automaton is 1-definite iff for every input letter $\sigma \in \Sigma$, there exists a state $q \in Q$ such that for every state $q' \in Q$, $\delta(q', \sigma)$ is either $\emptyset$ or q.*

The notion of definiteness is relaxed to characterize fusion-closed languages in terms of automata.

Definition 5. An automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi\rangle$ is *half definite* iff for every letter $\sigma \in \Sigma$ the states in $\{q' \mid \exists q \in Q\,.\,q' \in \delta(q,\sigma)\}$ are language equivalent.

Lemma 3. *If an ω-regular language $W \subseteq \Sigma^\omega$ is fusion-closed, then all deterministic automata that accept it are half-definite. If an ω-regular language $W \subseteq \Sigma^\omega$ is accepted by a half-definite deterministic automaton, then it is fusion-closed.*

Corollary 1. *An ω-regular language $W \subseteq \Sigma^\omega$ is fusion-closed and limit-closed iff it is accepted by a 1-definite safety automaton.*

The following theorem characterizes the ω-regular languages that are $\mathcal{R}$-generable in terms of the structure of their accepting automata. This provides an efficient membership test for safety languages that can be generated by relations.

Theorem 1. *A language $W \subseteq \Sigma^\omega$ is $\mathcal{R}$-generable iff it is accepted by an initially-free, 1-definite safety automaton.*

Proof: If a set $W \subseteq \Sigma^\omega$ is generated by a relation R, an initially-free, 1-definite safety automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, Q\rangle$ can be built as follows. For each letter $\sigma \in \Sigma$ that appears in some pair of R, a state q_σ is added to Q, distinct from q_{in}. Let $\delta(q_\sigma, \sigma') = q_{\sigma'}$ for each pair $(\sigma, \sigma') \in R$. Moreover, let $\delta(q_{\text{in}}, \sigma) = q_\sigma$ for every letter σ that appears in first position in some pair of R. This guarantees that A is initially-free because q_{in} is connected to every state in $Q \setminus q_{\text{in}}$ that has at least one outgoing transition. Then, A accepts W.

If $A = \langle \Sigma, Q, q_{\text{in}}, \delta, Q\rangle$ is an initially-free, 1-definite safety automaton accepting W, a relation R is built as follows. The pair of letters (σ, σ') is added to R when $\delta(q, \sigma') = q'$ and σ is a letter that labels the transitions into q. (No pair is added to R for a state with no incoming transitions.) Then, R generates W. ∎

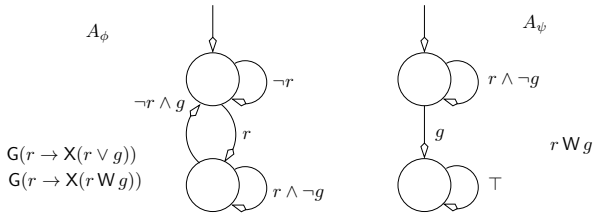


Fig. 1. Irredundant automata for three LTL formulae. All states have priority 1. A_ϕ is 1-definite and shows that both formulae represent transition constraints (transition constraints are defined in Sec. IV). A_ψ is not 1-definite because the input word $(r \wedge \neg g)^\omega$ cannot distinguish the target state.

The check of Theorem 1 can be simplified when the safety automaton is known to be deterministic and irredundant.

Lemma 4. *If a deterministic safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi\rangle$ is initially-free then it is also 1-definite.*

Example 1. Consider A_ϕ shown in Figure 1. It is a reduced, deterministic, initially-free, and 1-definite automaton. The language $L(A_\phi)$ is $\mathcal{R}$-generable and the relation R is given by the Boolean formula $(\neg r \wedge \neg r') \vee (\neg r \vee r') \vee (r \wedge r') \vee (r \wedge \neg r' \wedge g')$, which can be simplified to $\neg r \vee r' \vee g'$.

If a language $W \subseteq \Sigma^\omega$ is fusion-closed and limit-closed then it is a subset of some $\mathcal{R}$-generable language $W' \subseteq \Sigma^\omega$. For $R_{\text{in}} \subseteq \Sigma$, let $L(R_{\text{in}})$ be $R_{\text{in}}\Sigma^\omega$.

Theorem 2. *Given a fusion-closed and limit-closed language $W \subseteq \Sigma^\omega$, let $R_{in} = \{\sigma \mid \exists \sigma w \in W\}$. Then there exists an $\mathcal{R}$-generable language W' such that $W = W' \cap L(R_{in})$.*

An acceptor for the language W' in Theorem 2 is obtained through the construction in the proof of Theorem 1.

The following result is already foreshadowed in [17]. This theorem provides us with a method to detect ω-regular properties which are $\mathcal{R}$-generable.

Theorem 3. *If an irredundant safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi\rangle$ that accepts the ω-regular safety property φ is not 1-definite, then no other automaton that accepts φ is 1-definite.*

IV. LTL AND $\mathcal{R}$-GENERABILITY

This section provides a syntactic characterization of a subset of the LTL formulae that describe $\mathcal{R}$-generable languages. When a formula is syntactically $\mathcal{R}$-generable, the automata-based procedure of Sec. III can be skipped.

Definition 6. An LTL formula ϕ is a *transition-constraint* if it belongs to the class defined by the following grammar, in which x is a proposition:

$$P ::= \mathsf{G}(f), \qquad f ::= p \mid n \mid f \wedge f \mid f \vee f,$$
$$p ::= x \mid \neg x, \qquad n ::= \mathsf{X}\, p \ .$$

The grammar defines LTL formulae in negation normal form. Only the X operator is permissible inside the G operator and its nesting is not allowed. The set of safety properties described by Definition 6 is closed under conjunction.

Given a formula ϕ produced by the grammar in Definition 6, the relation R that generates the language of ϕ is obtained by replacing each subformula $\mathsf{X}\, x$ by x' and each subformula $\mathsf{X} \neg x$ by $\neg x'$. Finally, the G operator is discarded to obtain the propositional formula that is the representation of R. Conversely, given a relation $R \subseteq \Sigma \times \Sigma'$, an LTL safety formula ϕ_R in the form described in Definition 6 can be obtained by replacing each x' by $\mathsf{X}\, x$ and $\neg x'$ by $\neg \mathsf{X}\, x$, finally applying the G operator. (The relation $R \subseteq \Sigma \times \Sigma$ generated from a transition constraint is a Boolean formula, a minterm that satisfies this formula describes a pair $(\sigma, \sigma') \in R$ such that the cube of non-primed variables extracted from the minterm encodes σ and the cube of primed variables extracted from the minterm encodes σ'.)

Example 2. Some LTL formulae describe transition constraints even though they are do not satisfy Definition 6. Simple rewriting suffices for something like $\phi = \mathsf{G}(r \rightarrow \mathsf{X}(r \vee g))$, while the construction of an irredundant safety automaton is used to show that $\psi = \mathsf{G}(r \rightarrow \mathsf{X}(r\, \mathsf{W}\, g))$ describes the transition constraint ϕ. The formula $r\, \mathsf{W}\, g$ does not describe a transition constraint. The automaton for this property is shown on the right in Figure 1; this automaton is not 1-definite, but it is deterministic and irredundant. By Theorem 3, it does not accept an $\mathcal{R}$-generable language.

V. General Safety Properties

Safety properties like $r\, \mathsf{W}\, g$ are neither suffix-closed nor fusion-closed. The objective of this section is to find an $\mathcal{R}$-generable language $\hat{W}$ that embeds an arbitrary safety language W. It is shown that a fusion-closed and limit-closed language $\hat{W}$ over an augmented alphabet exists such that it is in one-to-one correspondence with W. Theorem 2 can be invoked to decompose $\hat{W}$ into the intersection of an $\mathcal{R}$-generable language and one that constrains the initial letter.

Given a safety language L that is not $\mathcal{R}$-generable, the problem of augmenting the alphabet Σ to $\hat{\Sigma} = \Sigma \times K$ is solved through the irredundant automaton A that accepts L.

Let $A_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$ be an irredundant deterministic safety automaton that accepts property ϕ. If A_ϕ is not 1-definite then there exists $\sigma \in \Sigma$ such that the automaton A_ϕ can be in two or more different states after reading the letter σ. This ambiguity of the irredundant automaton A_ϕ after reading one letter defines an incompatibility graph. The vertices of the graph are the states of the automaton, and there is an edge between two distinct vertices v_1 and v_2 iff there is a letter $\sigma \in \Sigma$ such that, for some states t_1 and t_2, $\delta(t_1, \sigma) = v_1$ and $\delta(t_2, \sigma) = v_2$. The chromatic number of this graph gives the minimum cardinality of the K required to turn the irredundant automaton into a 1-definite automaton. Each element of K corresponds to one of the colors and $\gamma : Q \rightarrow K$ maps states to colors. One can obtain another safety automaton $\hat{A}_\phi = \langle \hat{\Sigma}, Q, q_{\text{in}}, \hat{\delta}, \pi \rangle$, where $\hat{\Sigma} = \Sigma \times K$ and $\hat{\delta}(q, (\sigma, \gamma(\delta(q, \sigma)))) = \delta(q, \sigma)$. The label of each transition is augmented with the color of the target state; this guarantees that $\hat{A}_\phi$ is a 1-definite safety automaton. (If A_ϕ can be in several different states after reading a letter $\sigma \in \Sigma$ then all the states are colored differently in the incompatibility graph.) Since $\hat{A}_\phi$ is 1-definite, the transition function $\hat{\delta}$ can be replaced by a new transition function $\tilde{\delta} : \hat{\Sigma} \rightarrow Q$ where $\tilde{\delta}((\sigma, k)) = \{q' \mid \exists q \in Q \,.\, q' = \hat{\delta}(q, \sigma) \wedge k = \gamma(q')\}$. The state coloring function γ can also be replaced by an edge coloring function $\tilde{\gamma} : Q \times \hat{\Sigma} \rightarrow K$, where $\tilde{\gamma}(q, \sigma) = \gamma(\delta(q, \sigma))$. (The color of an initial state that does not have any incoming transitions is not important.) It will be seen in Sec. VI that the map $\tilde{\gamma}$ is convenient when checking realizability of the property ϕ through its transition constraint.

Example 3. The irredundant automaton A_{ϕ_1} for the property $\phi_1 = r\, \mathsf{W}\, g$ is shown in Figure 2. This is not a fusion-closed language, therefore no 1-definite automaton exists. Because of the input $r \wedge \neg g$ the two states are incompatible. The chromatic number of the incompatibility graph derived from A_{ϕ_1} is 2.

The automaton $\hat{A}_\phi$ derived from an irredundant A_ϕ through the coloring procedure described earlier accepts a fusion-closed and limit-closed language over the alphabet $\hat{\Sigma}$. The language of $\hat{A}_\phi$ can be represented by a relation R_ϕ and an initial predicate R_{in}. Let $\zeta : \hat{\Sigma} \rightarrow \Sigma$ be the projection function that maps letter $(\sigma, k) \in \hat{\Sigma}$ to σ; let $\zeta(w)$ and $\zeta(W)$ denote the point-wise extensions of ζ to a word $w \in \hat{\Sigma}^\omega$ and to a language $W \subseteq \hat{\Sigma}^\omega$. Then the language of $\hat{A}_\phi$ embeds the language described by ϕ so that $\zeta(L(\hat{A}_\phi)) = L(\phi)$.

The following lemma shows that the safety language accepted by A_ϕ is embedded in the language accepted by $\hat{A}_\phi$. It proves that every word in $L(A_\phi)$ has a corresponding word in $L(\hat{A}_\phi)$ through the runs of the automata A_ϕ and $\hat{A}_\phi$.

Lemma 5. *Given a safety property $W \subseteq \Sigma^\omega$, there exists an augmented alphabet $\hat{\Sigma}$ and an $\mathcal{R}$-generable language $\hat{W} \subseteq \hat{\Sigma}^\omega$ such that $\zeta : \hat{\Sigma}^\omega \rightarrow \Sigma^\omega$ is a bijection from $\hat{W}$ to W.*

Proof: Let A_ϕ be an irredundant safety automaton accepting W; let $\hat{A}_\phi$ be the 1-definite automaton obtained through the procedure described above. Let $\hat{W}$ be the language of $\hat{A}_\phi$. The automata A_ϕ and $\hat{A}_\phi$ are isomorphic and every edge (q, σ) of A_ϕ has a unique corresponding edge $(q, (\sigma, \tilde{\gamma}(q, \sigma)))$ in $\hat{A}_\phi$.

Therefore, for every word $w \in W$, there is a unique word $\hat{w} \in \hat{\Sigma}^\omega$ such that $\zeta(\hat{w}) = w$ and $\hat{w}$ has a run in $\hat{A}_\phi$. This run $\hat{\rho}$ is identical to the run ρ of w in A_ϕ. Hence, $\hat{w} \in \hat{W}$. Since, there is an injection from $\hat{W}$ to $\hat{\Sigma}$, the restriction of the function ζ to $\hat{W} \subseteq \hat{\Sigma}$ is a bijection from $\hat{W}$ to W. ■

Example 4. Continuing Example 3. The safety property $\phi_1 = r\, \mathsf{W}\, g$ is defined over the alphabet $\Sigma = 2^{\{r,g\}}$. Let $K = \{\neg x, x\}$ as $|K| = 2$. The irredundant 1-definite automaton $\hat{A}_{\phi_1}$ is shown in Figure 2. The relation R_{ϕ_1} is

$$((r \wedge \neg g \wedge \neg x) \wedge ((r' \wedge \neg g' \wedge \neg x') \vee (g' \wedge x'))) \vee (x')$$

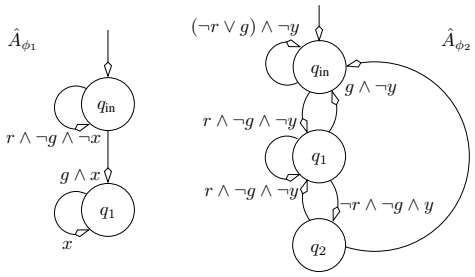


Fig. 2. Automata for $\phi_1 = r \,\mathsf{W}\, g$ and $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g)))$

and R_{in} is $((r \wedge \neg g \wedge \neg x) \vee (g \wedge x))$.

Example 5. The LTL formula $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g))$ over $\Sigma = 2^{\{r,g\}}$ does not describe a fusion-closed language. States q_{in} and q_2 are incompatible with each other. Let $K = \{\neg y, y\}$ as $|K| = 2$. The automaton $\hat{A}_{\phi_2}$ is shown in Figure 2. The predicate R_{in} is $\neg y$ and the relation R_{ϕ_2} is

$$((\neg r \wedge \neg g \wedge y \wedge ((r' \vee g') \wedge \neg y')) \vee ((\neg r \vee g) \wedge \neg y \wedge \neg y')) \\ \vee (r \wedge \neg g \wedge \neg y \wedge (((r' \vee g') \wedge \neg y') \vee (\neg r' \wedge \neg g' \wedge y'))) \ .$$

There exists another approach that can derive a transition constraint from an arbitrary safety property. A transition constraint can be derived from an LTL safety property by putting it in *separated normal form* [2]. For instance, $\mathsf{G}(r \rightarrow \mathsf{X}(r \,\mathsf{W}\, g))$ can be written as $\mathsf{G}((r \rightarrow \mathsf{X}\, x_1) \wedge (x_1 \rightarrow g \vee (r \wedge \mathsf{X}\, x_1)))$. This rewriting, however, may use more auxiliary variables than the approach based on the incompatibility graph.

Of course, given a conjunction of safety properties, obtaining transition constraints from each of them in turn and then conjoining all the transition constraints does not guarantee optimality. This is because the product automaton obtained from composing the irredundant automata for the corresponding safety properties may be neither reduced nor irredundant. In fact, the conjunction of two languages that are not fusion-closed and limit-closed may result in a fusion-closed and limit-closed (and maybe even suffix-closed) language.

VI. Realizability of Transition Constraints

This section describes how to check the realizability of a safety property ϕ embedded in a fusion-closed, limit-closed language $\hat{W}$. The language $\hat{W}$ is described by an initial predicate R_{in} and a relation R_ϕ. One can obtain an input-based game G_ϕ from the automaton A_ϕ that recognizes the language described by ϕ. One can also derive an input-based game $\hat{G}_\phi$ from the automaton that accepts $\hat{W}$. Finally a game $\hat{G}_\phi^{\mathcal{R}}$ can be derived from R_ϕ and R_{in}. It can be shown that one can obtain system's or environment's winning strategy for G_ϕ by playing $\hat{G}_\phi$ or vice-versa. Moreover, one can obtain system's or environment's winning strategy for $\hat{G}_\phi$ by playing $\hat{G}_\phi^{\mathcal{R}}$ or vice-versa. Therefore, one can obtain system's or environment's winning strategy for G_ϕ by playing $\hat{G}_\phi^{\mathcal{R}}$.

For lack of space, the details of the constructions of strategies for one game from those of the other are omitted. (These constructions are rather lengthy and tedious and are not used in the synthesis process: they are only used to prove its correctness.) However, it must be mentioned that in $\hat{G}_\phi^{\mathcal{R}}$ the input letter includes the "color" added to A_ϕ to obtain $\hat{A}_\phi$. The choice of the color is given to the system. Since there is only one way to choose the right color, and the system needs to make the right choice to win, this additional responsibility does not affect the outcome of the game.

Now it is shown how to check for the existence of winning strategies in $\hat{G}_\phi^{\mathcal{R}}$ symbolically; that is, by an algorithm that manipulates the characteristic functions of sets and relations over $\hat{\Sigma}$. The game $\hat{G}_\phi^{\mathcal{R}}$ is played in two stages; the first stage checks the realizability of R_ϕ and the second stage checks the realizability of $R_{\text{in}} \wedge R_\phi$. Given a set of target letters $T(X')$, that is, a set expressed in terms of next-state variables, the pre-image operator[3] MX is defined as follows:

$$\mathsf{MX}_\phi\, T = \forall X'_{ed} \,.\, \exists X'_s \,.\, \forall X'_{ep} \,.\, \exists X'_K \,.\, R_\phi(X, X') \wedge T(X') \ .$$

The greatest fixpoint operator MG_ϕ is defined as $\mathsf{MG}_\phi\, p = \nu Z \,.\, p \wedge \mathsf{MX}_\phi\, Z$. The realizability of R_ϕ is checked by computing the *realizable subset* $\hat{\Sigma}_r^\phi$ of $\hat{\Sigma}$ such that $\hat{\Sigma}_r^\phi = \mathsf{MG}_\phi\, \top$. The greatest fixpoint computation removes the *terminal letters* from the alphabet $\hat{\Sigma}$. The terminal letters of the alphabet are defined inductively as letters after which there does not exist a strategy to pick a next letter such that R_ϕ is satisfied or letters after which only terminal letters can be selected to satisfy R_ϕ. Finally, the realizability of $R_{\text{in}} \wedge R_\phi$ is checked. The system wins the game $\hat{G}_\phi^{\mathcal{R}}$ iff $\mathsf{MX}_{\text{in}}\, \mathsf{MG}_\phi\, \top = \top$, where

$$\mathsf{MX}_{\text{in}}\, T = \forall X'_{ed} \,.\, \exists X'_s \,.\, \forall X'_{ep} \,.\, \exists X'_K \,.\, R_{\text{in}}(X') \wedge T(X') \ .$$

For every letter in $\hat{\Sigma}_r^\phi$, the system can always pick the next letter from the same set so that R_ϕ is satisfied. The operator MX_{in} establishes the system's ability to start a word from a letter in $\hat{\Sigma}_r^\phi$ such that R_{in} is satisfied. Therefore, the system wins the game $\hat{G}_\phi^{\mathcal{R}}$ iff $\mathsf{MX}_{\text{in}}\, \mathsf{MG}_\phi\, \top = \top$, which means that the system can force the selection of a letter from $\hat{\Sigma}_r^\phi$ ($\hat{\Sigma}_r^\phi = \mathsf{MG}_\phi\, \top$) such that the predicate R_{in} is also satisfied.

Example 6. Examples 4 and 5 are continued here. Consider the property $\phi = \phi_1 \wedge \phi_2$ where $\phi_1 = r \,\mathsf{W}\, g$ and $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g)))$. Let $X_{ed} = \emptyset$, $X_s = \{r\}$, $X_{ep} = \{g\}$, and $X_K = \{x, y\}$. Initially r is asserted until g is asserted; after that, whenever r is asserted then it must be reasserted at least every other step until g is asserted. The iterates of the $\mathsf{MG}_\phi\, \top$ computation are $Z_0 = \top$, $Z_1 = (x \wedge \neg y) \vee (x \wedge \neg r \wedge \neg g) \vee (\neg y \wedge r \wedge \neg g)$, $\hat{\Sigma}_r^\phi = Z_2 = Z_1$. Since $\mathsf{MX}_{\text{in}}\, Z_2 = \top$, property ϕ is realizable.

VII. Synthesis from Transition Constraints

This section reviews Boolean equations [4] and their relation to safety games. In particular, the connection between the solution of Boolean equations and the winning strategy of the safety game is established. The synthesis approach discussed in this section scales well when the specification contains a large percentage of safety properties.

[3] In Sec. VIII the environment has to check the realizability of an assumption ϕ. In that case the pre-image operator is $\mathsf{MX}_\phi\, T = \exists X'_{ed} \,.\, \forall X'_s \,.\, \exists X'_{ep} \,.\, \exists X'_K \,.\, R_\phi(X, X') \wedge T(X')$.

A. Boolean Equations

Let $x_1, \ldots, x_m$ and $y_i, \ldots, y_n$ be two sets of variables ranging over a Boolean algebra B. ($\neg$, $\vee$, and $\wedge$ denote complementation, join, and meet in B, respectively.) A Boolean equation in independent variables $x_1, \ldots, x_m$ and unknowns $y_1, \ldots, y_n$ is a formula of the form

$$\forall x_1.., x_m.\exists y_1.., y_n.F_0(x_1.., x_m)=F(y_1.., y_n, x_1.., x_m), \quad (1)$$

where $F_0 = \exists y_1, \ldots, y_n \,.\, F$ is the *consistency condition* of F. When no confusion arises, we write F to signify (1).

A *particular solution* of (1) is a set of Boolean functions $f_i(x_1, \ldots, x_m)$, for $1 \leq i \leq n$, such that

$$\forall x_1.., x_m \,.\, F_0(x_1.., x_m) = F(f_1.., f_n, x_1.., x_m) \ .$$

A *general solution in parametric form* of (1) is a set of Boolean functions $g_i(x_1, \ldots, x_m,\, p_1, \ldots, p_i)$, for $1 \leq i \leq n$, where each p_j is a Boolean function of $x_1, \ldots, x_m$, such that

$$\forall p_1.., p_n \,.\, \forall x_1.., x_m \,.\, F_0(x_1.., x_m) = F(g_1.., g_n, x_1.., x_m) \ ,$$

and for every particular solution $\{f_1, \ldots, f_n\}$ of (1) there is a choice of p_j's that produces a particular solution $\{f'_1, \ldots, f'_n\}$ such that, for $1 \leq i \leq n$,

$$\forall x_1.., x_m \,.\, F_0(x_1.., x_m) \leq f_i(x_1.., x_m) \leftrightarrow f'_i(x_1.., x_m) \ .$$

A general solution to (1) can be computed by the method of *successive eliminations* [4], which, given F, returns F_0 and the solution functions g_i. Letting $F_n = F$ and $F_{i-1} = \exists y_i \,.\, F_i$ for $1 \leq i \leq n$, it produces

$$\begin{aligned} g_i = \neg F_i(g_1.., g_{i-1}, \bot, x_1.., x_m) \vee \\ (p_i \wedge F_i(g_1.., g_{i-1}, \top, x_1.., x_m)) \ . \end{aligned} \quad (2)$$

Example 7. Consider $F_2 = (\neg x_1 \wedge y_1) \vee (x_2 \wedge y_2)$. Then

$$\begin{aligned} F_1{=}\exists y_2.F_2{=}(\neg x_1 \wedge y_1) \vee x_2 \quad F_0{=}\exists y_1.F_1{=}\neg x_1 \vee x_2 \\ g_1(x_1, x_2, p_1) = \neg x_2 \vee (p_1 \wedge \neg x_1) \vee (p_1 \wedge x_2) \\ g_2(x_1, x_2, p_1, p_2) = x_1 \vee (x_2 \wedge \neg p_1) \vee p_2 \ . \end{aligned}$$

One can verify that $\forall p_1, p_2, x_1, x_2.\neg x_1 \vee x_2{=}F(g_1, g_2, x_1, x_2)$.

Setting $p_1 = p_2 = \bot$ in g_1 and g_2, one obtains the particular solution $f_1 = \neg x_2$, $f_2 = x_1 \vee x_2$. The same solution is obtained for $p_1 = \neg x_2$ and $p_2 = x_1 \vee x_2$. The particular solution $f'_1 = \neg x_2$, $f'_2 = x_2$ cannot be obtained from g_1 and g_2, but, for $i \in \{1, 2\}$, $\forall x_1, x_2 \,.\, \neg x_1 \vee x_2 \leq f_i \leftrightarrow f'_i$. Note that when the consistency condition is identically satisfied,

$$\neg F_i(g_1.., g_{i-1}, \bot, x_1.., x_m) \leq F_i(g_1.., g_{i-1}, \top, x_1.., x_m)) \ .$$

Therefore, if p_i is taken in the interval defined by the two bounds, the particular solution obtained for y_i is p_i itself.

Solving a Boolean equation can be interpreted as finding winning strategies for a two-player game. One player selects a value $(\hat{x}_1, \ldots, \hat{x}_m) \in (\{\bot, \top\})^m$ for the independent variables, while the other must choose a value $(\hat{y}_1, \ldots, \hat{y}_n) \in (\{\bot, \top\})^n$ for the unknowns such that $F_0(\hat{x}_1, \ldots, \hat{x}_m) = F(\hat{y}_1, \ldots, \hat{y}_n,\, \hat{x}_1, \ldots, \hat{x}_m)$. A particular solution to the equation gives one winning strategy for the second player, while a general solution describes all winning strategies (that differ over the consistency condition).

B. Parameterized Solutions and Transition Constraints

Once the realizability of an ω-regular safety property ϕ is established, an implementation that satisfies ϕ can be generated from a system's winning strategy in the game $\hat{G}^{\mathcal{R}}_{\phi}$. This section describes the procedure to obtain an implementation that satisfies ϕ from the initial predicate R_{in} and relation R_ϕ. The solution of equations derived from R_{in} defines the initial condition, while the solution of equations derived from R_ϕ defines the steady state behavior.

The parameterized reactive system M_ϕ that implements ϕ consists of the solutions for the initial values and steady state values for variables in $X'_s \cup X'_K$ and a state variable $\mathcal{I}$ which is initially $\bot$ and then is $\top$ forever. For each element $u' \in (X'_s \cup X'_K)$, let u_{in} be the initial solution and u_∞ the steady-state solution. The initial value of u is $\bot$ and its update is given by $u' = (\neg\mathcal{I} \wedge u'_{\text{in}}) \vee (\mathcal{I} \wedge u'_\infty)$. The initialization bit distinguishes between the initial and steady state solutions.

We now describe how the solutions for initial values and steady state values are computed. The relation R_ϕ is defined over the variables $\hat{X}$ and $\hat{X}'$, where $\hat{X} = X_{ed} \cup X_s \cup X_{ep} \cup X_K$, while $\hat{X}' = X'_{ed} \cup X'_s \cup X'_{ep} \cup X'_K$. Given $\hat{\Sigma}^\phi_r = \mathsf{MG}_\phi \top$ the following four relations are used to synthesize an implementation for the property ϕ:

$$\begin{aligned} F &= R_\phi(\hat{X}, \hat{X}') \wedge \hat{\Sigma}^\phi_r(\hat{X}') & F_s &= \forall X'_{ep} \,.\, \exists X'_K \,.\, F \\ I &= R_{\text{in}}(\hat{X}') \wedge \Sigma^\phi_r(\hat{X}') & I_s &= \forall X'_{ep} \,.\, \exists X'_K \,.\, I \ . \end{aligned}$$

The existence of solutions of these Boolean equations has been established by checking the realizability of ϕ through R_{in} and R_ϕ. The steady state solution for the variables in X'_s is computed from F_s. The steady state solution for the variables in X'_K is computed from F. The solution for the initial values for variables in X'_s is computed from I_s. The solution for the initial values for variables in X'_K is computed from I. If a variable X'_s appears in the steady-state (initial) solution of X'_K then it is substituted by its steady-state (initial) solution.

Example 8. Continuing Example 6, a system M_ϕ is obtained through the synthesis of x', y' and r'. The steady state Boolean equation for the unknown variables $\{x', y'\}$ is $F_4 = R_\phi(\hat{X}, \hat{X}') \wedge \hat{\Sigma}^\phi_r(\hat{X}')$. This equation can be computed from the values of R_ϕ and Σ^ϕ_r described in Example 6. Let $\{r_i, x_i, y_i\}$ be the set of parameters, then the steady state solution of variables $\{x', y'\}$ is given by:

$$\begin{aligned} y'_\infty &= \neg F_3^{\neg y'} \vee (y_i \wedge F_3^{y'}) = \ r \wedge \neg g \wedge \neg p_1 \wedge x \wedge \neg g' \ , \\ x'_\infty &= \neg F_4^{\neg x'} \vee (x_i \wedge F_4^{x'}) = \ \neg r \vee g \vee x \vee g' \ . \end{aligned}$$

If the variable y' had appeared in x'_∞ then y' would be substituted by its function y'_∞. The steady state solution of the variables in $\{r'\}$ is computed from the Boolean equation $F_1 = \forall g' \,.\, \exists\{x', y'\} \,.\, F_4$, where

$$r'_\infty = \neg F_1^{\neg r'} \vee (r_i \wedge F_1^{r'}) = \ p_1 \vee \neg x \vee y \ ,$$

Likewise the initial values for $\{x', y'\}$ are synthesized from $I = ((r' \wedge \neg g' \wedge \neg x') \vee (g' \wedge x')) \wedge \neg y'$, where

$$y'_{\text{in}}{=}\neg I_3(0) \vee \big(y_i \wedge I_3(1)\big){=}\bot, x'_{\text{in}}{=}\neg I(0) \vee \big(x_i \wedge I(1)\big){=}g'.$$

The initial value of r' is computed from $I_1 = \forall X'_{ep} \,.\, \exists X'_K \,.\, I$, where $r'_{\text{in}} = \neg I_1(0) \vee \big((r_i \wedge I_1(1)\big) = \top$.

Each variable $v \in \hat{X}$ represents a latch (register) which stores the current value of the corresponding value of $v' \in \hat{X}'$. Each variable $o' \in X'_s$ represents the output of the sequential machine and is labeled as the corresponding variable o. Each variable $o' \in X'_K$ is stored in the latch represented by the variables in X_K, these are treated as internal signals.

The solution is kept in parameterized form so that winning strategies for the progress properties can be found. This is done by computing the appropriate values of the parameters (which may be functions requiring some finite memory to satisfy the progress properties). If the specification does not contain any progress properties then a simplified M_ϕ can be obtained by assigning any values to the parameters.

VIII. Obligation Properties

If a game with an ω-regular winning condition has a graph with more than one strongly connected component (SCC) then the winning and losing states can be computed inductively starting from the terminal SCCs. At each non-terminal SCC, one computes the states that each player can control to its winning states outside of the SCC (which are already known). The game is then played on the remaining states. This approach is discussed in [12]. In this section this idea is applied to the obligation properties defined by the implication of two safety properties (e.g., environment assumption and system guarantee). Every obligation property results in a DPW of minimum index 1 with more than one SCC.

To check realizability of an implication between two safety properties such as $\psi \rightarrow \phi$, one converts ψ and ϕ to parity games G_ψ and G_ϕ with safety conditions π_ψ and π_ϕ. The SCCs of their product can be partitioned in three ordered sets; the bottom set S_B contains the states in which the antecedent has been violated. The middle set S_M contains the states in which only the consequent has been violated, and the top set S_T contains the states where both properties hold. The states in $S_T \cup S_B$ have priority 1; those in S_M have priority 0.

This game does not need to be built explicitly, though. Given an implication $\Phi = \psi \rightarrow \phi$, where both ψ and ϕ are safety properties, one can obtain the relations R_ψ and R_ϕ as described in Sec. V. These relations are used to check the realizability of Φ. The pre-image operator MX_ϕ defined in Sec. VI cannot be used for checking Φ because it computes the states that can be forced by the protagonist to stay within the SCC, while in a game obtained from Φ, the protagonist may be able to win the game by staying within S_T or by forcing a move out of S_T to states from which it can force the play to S_B. Therefore, a modified pre-image operator needs to be defined that takes into account the protagonist's option to escape the SCC. For lack of space we only outline its use.

The solution of the game obtained from Φ follows three steps. In the first step one plays the game $\mathsf{MG}_\psi \top$ to compute the letters from which the environment can satisfy R_ψ. In this game the environment is the protagonist and the system is the antagonist. One may need to augment the alphabet Σ to $\tilde{\Sigma} = \Sigma \times K_e$ as described in Sec. V; the control of coloring variables X_{K_e} is assigned to the environment. The system is eventually able to force a violation of R_ψ from the letters in $\tilde{\Sigma} \backslash \Sigma_r^\psi$. From the letters Σ_r^ψ, the system can only satisfy R_Φ by satisfying R_ϕ. The new pre-image operator is used to compute $\mathsf{MG}_\Phi \top$; the objective of the system is to keep satisfying R_ϕ while the environment cannot use strategies that will give the system the option to choose the letter from $\tilde{\Sigma} \backslash \Sigma_r^\psi$. Once again, one may need to augment the alphabet $\tilde{\Sigma}$ to $\hat{\Sigma} = \tilde{\Sigma} \times K_s$. The system is able to satisfy R_Φ from the letters in $\Sigma_r^\Phi = \mathsf{MG}_\Phi \top \cup (\hat{\Sigma} \setminus (\Sigma_r^\psi \times K_s))$. Finally, the system wins the game obtained from Φ when the constraint $R_{\text{in}}^\psi \rightarrow R_{\text{in}}^\phi$ allows the system to select a letter from Σ_r^Φ.

As discussed in Sec. V, the control of coloring variables is assigned to the player who is trying to satisfy the property. This is why the variables in X_{K_e} (X_{K_s}) are controlled by the environment (system) when it is trying to satisfy ψ (ϕ). This distinction is at work in the following example.

Example 9. Consider the LTL formula $\Phi = \psi \rightarrow \phi$, where $\psi = \mathsf{G}(r \wedge \mathsf{X}\, r \rightarrow \mathsf{XX}(r \rightarrow l))$ and $\phi = \mathsf{G}((r \wedge \neg l \rightarrow \neg g) \wedge (r \wedge \mathsf{X}\, r \rightarrow \mathsf{XX}(r \rightarrow g)))$. Then R_{in}^ψ is $\neg x$ and R_ψ is

$$(\neg r \rightarrow \neg x') \vee (r \wedge \neg x \rightarrow (r' \leftrightarrow x')) \vee (r \wedge x \rightarrow ((r' \wedge l' \wedge x') \vee (\neg r' \wedge \neg x'))) \ ,$$

while R_{in}^ϕ is $(\neg r \vee l \vee \neg g) \wedge \neg y$ and R_ϕ is

$$(r \wedge \neg l \rightarrow \neg g) \wedge \big((\neg r \rightarrow \neg y') \vee (r \wedge \neg y \rightarrow (r' \leftrightarrow y')) \vee (r \wedge y \rightarrow ((r' \wedge g' \wedge y') \vee (\neg r' \wedge \neg y')))\big) \ .$$

The alphabets $\Sigma_\Phi = 2^{\{r,l,g\}}$ and $\hat{\Sigma}_\Phi = 2^{\{r,l,g,x,y\}}$, where

$$X_{ed}{=}\{r,l\},\ X_s{=}\{g\},\ X_{ep}{=}\emptyset,\ X_{K_e}{=}\{x\},\ X_{K_s}{=}\{y\}.$$

The system loses both games $G_{\neg\psi}$ and G_ϕ, but it can win the game G_Φ. In the game G_ϕ the environment can force the system to violate ϕ at any time by playing the sequence $r \wedge \neg l, r \wedge \neg l, r \wedge \neg l$. On the other hand, in the game G_Φ, this sequence forces the environment to violate ψ and if the environment never plays this sequence then system can always satisfy ϕ. Thus G_Φ is won by the system.

IX. Experimental Results

The approach described here has been implemented in Vis [3] as an extension of the Safety-First approach described in [21]. We report on preliminary experiments conducted on a parameterized example coming from [1] (Amba Bus). The performance and quality of the implementation is compared in Table I to Anzu [1] and our Safety-First approach.

The specifications used for Anzu are different from those used by the other two tools because Anzu requires the safety components to be pre-synthesized into transition constraints. The salient feature of our approach is the significantly smaller sizes of the implementation. We only report latch count in [21], but starting from a more abstract specification than Anzu was paid with higher latch and gate counts. The new approach, however, keeps the simpler and more abstract specification, and still manages to achieve the most efficient designs.

The specification of the Amba bus controller for n clients contains $n-1$ properties that are $\mathcal{R}$-generable, but are not

TABLE I. Experimental Results

Model	Safety		Parity		Properties	Time(s)			latches			Gates	
	E	S	E	S	ANZU	ANZU	SF	SF+TC	ANZU	SF	SF+TC	ANZU	SF+TC
AMBA2	3	17	2	3	56	2.39	6.87	0.44	24	37	15	4409	281
AMBA3	4	22	2	4	68	44.67	14.2	1.25	30	42	18	20686	586
AMBA4	5	26	2	5	80	35.30	109.9	3.47	34	48	20	17501	860
AMBA5	6	31	2	6	93	224.06	139.7	5.23	39	56	22	48154	1747
AMBA6	7	34	2	7	105	1011.7	301.1	10.18	43	55	23	74948	1792
AMBA7	8	38	2	8	117	1758.5	965.6	17.93	48	61	24	88808	1714
AMBA8	9	41	2	9	129	2034.9	875.3	76.72	52	67	26	222598	3621
AMBA9	10	44	2	10	141	7861.2	1439.6	193.90	57	77	28	175298	4597
AMBA10	11	48	2	11	153	28319.8	3727.6	224.21	61	81	29	172195	4941
AMBA11	12	51	2	12	165	8403.3	3154.0	410.44	65	87	30	179291	6240
AMBA12	13	55	2	13	177	49138.7	6641.2	878.63	69	92	31	224266	7223
AMBA13	14	60	2	14	189	13163.4	32562.4	1335.04	73	98	32	239494	9361
AMBA14	15	64	2	15	200	17104.9	12202.2	1865.70	77	105	33	284027	8202
AMBA15	16	69	2	16	212	TO	TO	2611.00	-	-	34	-	12385

produced by the grammar in Definition 6. The specification also contains two safety properties that are not $\mathcal{R}$-generable irrespective of the number of clients.

We also implemented a limited retiming step (not applicable in the SAFETY-FIRST approach), applied after the safety properties of the system have been synthesized. Consider the function $f = (a_L \wedge b_L) \vee f_i$, where a_L and b_L are the latched versions of the signals a and b. One can implement this function with one latch $c = a \wedge b$ then $f = c_L \vee f_i$. This step may reduce the number of memory elements in the parameterized representation of the transition function. Both the combinational and sequential logic is reduced by this step.

In the case of AMBA bus controller, retiming has significant impact because this controller can be implemented as a Moore machine. When a Moore implementation is not possible, the effectiveness of retiming may be less noticeable. The runtime of the algorithm is affected by retiming because the parameterized representation is also simplified: with fewer BDD variables finding suitable variable orders becomes easier.

The results of Table I do not make use of conversion of general safety properties to transition constraints. Rather the automata for these properties are used directly. In the case when general safety properties are converted to transition constraints, the extraction of an optimal parameterized transition relation incurs significant penalty and is being investigated.

X. CONCLUSION

We have presented a technique that obtains a significantly simpler representation of the synthesis game. This results in significant improvement in solving the game and produces implementations that are an order of magnitude smaller than previous techniques. The results being reported here include the logic that keeps track of the environment's assumptions. However, this logic is often not required after the game has been played. We are investigating techniques that will allow the extraction of an implementation that only depends on this logic when absolutely necessary.

REFERENCES

[1] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[2] A. Bolotov and M. Fisher. A resolution method for CTL branching time temporal logic. In *Fourth International Workshop on Temporal Representation and Reasoning (TIME)*. IEEE Press, 1997.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer, Boston, 1990.

[5] O. Carton and R. Maceiras. Computing the Rabin index of a parity automaton. *Theoretical Informatics and Applications*, 33:495–505, 1999.

[6] R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)*, pages 91–100, 2012.

[7] E. A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983.

[8] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, Oct. 1991.

[9] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 117–123, Miami, FL, Jan. 2006.

[10] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, second edition, 1978.

[11] L. H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3(4):376–384, 1969.

[12] M. Lange and O. Friedmann. The pgsolver collection of parity game solvers. Technical report, Ludwig-Maximilians-Universität - München, 2009.

[13] P. Madhusudan. Synthesizing reactive programs. In *CSL*, pages 428–442, 2011.

[14] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Annual ACM Symposium on Principles of Distributed Computing*, pages 377–410, Quebec City, Quebec, Canada, Aug. 1990.

[15] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[16] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Computation Theory*, pages 157–168. Springer-Verlag, 1984. LNCS 208.

[17] M. Perles, M. O. Rabin, and E. Shamir. The theory of definite automata. *IEEE Transactions on Electronic Computers*, pages 233–243, 1963.

[18] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science*, pages 255–264, Seattle, WA, Aug. 2006.

[19] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[20] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[21] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for the synthesis of reactive systems. *Software Tools for Technology Transfer (Online First)*, pages 1–22, 2012.

On the Feasibility of Automation for Bandwidth Allocation Problems in Data Centers

Yifei Yuan, Anduo Wang, Rajeev Alur, and Boon Thau Loo
University of Pennsylvania

***Abstract*—Mapping virtual networks to physical networks under bandwidth constraints is a key computational problem for the management of data centers. Recently proposed heuristic strategies for this problem work efficiently, but are not guaranteed to always find an allocation even when one exists. Given that the bandwidth allocation problem is NP-complete, and the state-of-the-art SAT solvers have recently been successfully applied to NP-hard problems in planning and formal verification, the goal of this paper is to study whether these SAT solvers can be used to solve the bandwidth allocation problem exactly with acceptable overhead. We investigate alternative ways of encoding the allocation problem, and develop techniques for abstraction and refinement of network graphs for scalability. We report experimental comparisons of the proposed encodings with the existing heuristics for typical data-center topologies.**

I. Introduction

Allocating computing resources to customers' requests is the central task for a data center provider. The requests submitted usually define virtual networks involving a number of virtual machines (VMs) and also the bandwidth requirement for virtual links between VMs. To handle such a request, the data center provider should allocate server resources, as well as bandwidth on the links of the data center's physical network.

However, bandwidth allocation is a computationally hard problem. It is NP-complete to determine whether there is a valid allocation even for simple topologies of physical networks and virtual networks. Current proposed techniques focus on designing heuristic algorithms processing requests efficiently ([1], [2], [3]). These heuristics only consider local constraints, e.g. if there is enough link bandwidth of a server to host a VM [1], to determine allocation. While these techniques work efficiently, they provide no guarantee to always succeed in finding an allocation even if one exists.

On the other hand, recent success of applying SAT/SMT solvers [4], that is, solvers for constraint satisfaction problems, to NP-hard problems in planning and formal verification suggests a promising way to solve NP-hard problems in practice ([5], [6], [7], [8]). In this paper, we encode the bandwidth allocation problem into SAT formulas and utilize state-of-the-art SAT/SMT solvers to find the feasible allocation. We develop abstraction and refinement for the physical networks for scalability. Our experimental results show that the SAT approaches work effectively with acceptable overhead for small data center networks.

This research was partially supported by the NSF Expeditions in Computing project ExCAPE (CCSF 1138996).

The remaining paper is organized as follows. Section II provides a formulation of the bandwidth allocation problem, and the hardness result of it when restricting to the topology of trees. In Section III, we show the SAT encoding of the bandwidth allocation problem and we develop abstraction and refinement techniques for scalability. In Section IV, we show the simulation results. Section V concludes this paper.

II. Problem Formulation

In this section, we provide a formal definition of the bandwidth allocation problem. Formally, we model the physical network of a data center as a graph $PN = (A \cup B, L)$, where A is the set of host servers, B is the set of switches. L is the set of physical links that connect servers(switches) with switches. Moreover, each host server $s \in A$ has a capacity $c(s)$ that models the maximum number of virtual machines which a server can host. Each physical link $l \in L$ has a bandwidth $b(l)$. The virtual network is also modeled as a graph $VN = (V, E)$, and V is the set of virtual machines, E is the set of virtual links connecting virtual machines. Each virtual link $e \in E$ has a bandwidth requirement $r(e)$. The mapping $f : V \to A$ maps a VM to a host server. Due to the capacity of servers, f should not map VMs more than a server's capacity. Let P denote the set of paths in PN. The mapping $\rho : E \to P$ defines the routing path for each virtual link. That is $\rho(v_1, v_2) = (f(v_1), \cdots, f(v_2))$. To meet the bandwidth requirement, every physical link on the routing path must have enough available bandwidth for the virtual link.

We define the *bandwidth allocation* problem as follows.

Given a physical network $PN = (A \cup B, L)$ with server capacity c and link bandwidth b, and a virtual network $VN = (V, E)$, the bandwidth allocation problem seeks to find the mapping functions f and ρ, satisfying the following two conditions: (1) $\forall s \in A$, $c(s) \geq |\{v \in V | f(v) = s\}|$, and (2) $\forall l \in L$, $b(l) \geq \sum_{e \in E : l \in \rho(e)} r(e)$.

A feasible allocation can be checked in polynomial time and thus the bandwidth allocation problem is in the class NP. It is also proved in [1] that the bandwidth allocation problem is NP-hard in a general physical network.

In data centers, the tree structure is widely deployed for the physical network topology. Even though the routing path in a tree between any two servers is unique, the bandwidth constraint is still the bottleneck for solving the problem efficiently. We show that even when restricting the physical network topology to be a tree, the bandwidth allocation problem is

ISBN 978-0-9835678-3-7/13.

still strongly NP-hard, i.e., NP-hard even when the numerical parameters are encoded in unary.

Theorem 1 (NP-hardness for Trees)**.** *Given a physical network PN in the tree structure, and a virtual network VN, finding the mappings f and ρ is strongly NP-complete.*

Proof. Membership in NP is evident. For the NP-hardness, we show the reduction from a strongly NP-complete problem, namely, the 3-partition problem: given a multiset S of $3m$ positive numbers $\{x_1, ..., x_{3m}\}$, partition S into m subsets $S_1, ...S_m$, such that each subset has 3 numbers and the sum of the numbers in each subset is equal. We construct the physical network as a 1-level tree with $m+1$ servers $s_1, ..., s_{m+1}$. Each server s_i connects to a switch s. For $i = 1, ..., m$, set the bandwidth of the link connecting s_i and s to be t, which is the desired sum of each subset in the 3-partition problem, and the bandwidth of link (s_{m+1}, s) to be $3t$, which is the sum of all the numbers. Let the capacity of each server be 3. We construct a virtual network as a 1-level tree with $3m+1$ virtual machines. Suppose the leaves are $v_1, .., v_{3m}$ and the root is v_{3m+1}. The bandwidth requirement for the virtual link (v_i, v_{3m+1}) is x_i. Since the virtual machine v_{3m+1} can only be mapped to the server s_{m+1}, it is evident that there is a one-to-one mapping between these two instances. Therefore, the bandwidth allocation problem for trees is also NP-hard. □

III. SAT Approach

In this section, we provide an alternative solution to the bandwidth allocation problem using SAT/SMT solvers. First, we show how to encode the bandwidth allocation problem into SAT formulas with integers. Second, we develop abstraction and refinement of physical network topologies for scalability.

A. SAT Encoding

In this section, we show how to encode a bandwidth allocation problem into SAT formulas that work for any physical network topology and virtual network topology. For each virtual machine $v \in V$, and server $s \in A$, let $X(v, s)$ be an integer variable indicating $f(v) = s$, i.e. $X(v, s) = 1$ if $f(v) = s$, and $X(v, s) = 0$, otherwise. To ensure that each virtual machine gets mapped, and only mapped to one server, we have the constraint that

$$\alpha_s : \bigwedge_{v \in V} \left(\sum_s X(v, s) = 1 \right).$$

To encode the routing path for each pair of virtual machines, we define the variable $R(l, e, k)$ to indicate that the physical link l is the k'th link of the routing path that is allocated to the virtual link e. The following formula encodes the constraint that there is no more than 1 physical link as the k'th one:

$$\alpha_r : \bigwedge_{e,k} \left(\sum_{l \in L} R(l, e, k) \leq 1 \right).$$

The following constraint ensures that $R(l, e, k)$ indeed encodes a path in the physical network.

$$\alpha_c : \bigwedge_{e,k} \left(\bigvee_{l_1, l_2 : l_1, l_2 \text{ are adjacents}} R(l_1, e, k) \wedge R(l_2, e, k+1) \right).$$

The constraint $\bigvee_{l_1, l_2 : l_1, l_2 \text{ are adjacents}} R(l_1, e, k) \wedge R(l_2, e, k+1)$ means that the k'th and the $k+1$'th physical link on the routing path assigned for the virtual link e should be adjacent. For each physical link $l \in L$ and virtual link $e \in E$, let $Y(l, e)$ be an integer variable indicating the bandwidth of l reserved for virtual link e. We have the following constraint:

$$\alpha_y : Y(l, e) = r(e) \Leftrightarrow \bigvee_k R(l, e, k) = 1.$$

To encode the constraint that for each virtual link, there is a routing path between the host servers to which the two VMs of the virtual link are mapped, we have:

$$\alpha_v : \bigwedge_{\substack{(v_1, v_2) \in E, \\ s_1, s_2 \in A, s_1 \neq s_2}} \Bigg((X(v_1, s_1) = 1 \wedge X(v_2, s_2) = 1) \rightarrow \bigvee_{\substack{l_1 : s_1 \in l_1 \\ l_2 : s_2 \in l_2}} (Y(l_1, e) = r(e) \wedge Y(l_2, e) = r(e)) \Bigg).$$

Finally, the server capacity condition can be encoded as

$$\beta_{server} : \bigwedge_{s \in A} \left(\sum_v X(v, s) \leq c(s) \right)$$

, and the link capacity condition is encoded as

$$\beta_{link} : \bigwedge_{l \in E} \left(\sum_e Y(l, e) \leq b(l) \right).$$

Putting all the pieces together, we have the encoding $\Phi_{PN,VN}$ for the bandwidth allocation problem:

$$\Phi_{PN,VN} = \alpha_s \wedge \alpha_r \wedge \alpha_c \wedge \alpha_y \wedge \alpha_v \wedge \beta_{server} \wedge \beta_{link}.$$

The following theorem easily follows.

Theorem 2. *Given any physical network $PN = (A \cup B, L)$ and virtual network $VN = (V, E)$, there exists mappings f and ρ satisfying the requirements of the bandwidth allocation problem, if and only if the formula $\Phi_{PN,VN}$ is satisfied.*

B. Abstraction and Refinement

We use 2-level tree as an example topology for the data centers to demonstrate how to abstract and refine the network topology. In the tree topology, all the leaves are host servers, and there are two levels of switches. On the first level, the switches connect with the host servers, and the root switch sitting on the second level connects with the switches on the first level. For notation brevity, let T denote the physical tree topology. We denote the root of the 2-level tree as r, and its children as $w_1, ..., w_k$, and $s_{i,j}$ as the jth children of w_i.

The routing path for any two servers in a tree is unique, therefore, the encoding involving $R(l, e, k)$ can be omitted.

For each pair of host servers u, v, let $p(u, v)$ be the unique path between them. The constraint α_v can be simplified as:

$$\alpha_v : \bigwedge_{\substack{(v_1,v_2)\in E,\\ s_1,s_2\in A, s_1\neq s_2}} \Big((X(v_1, s_1) = 1 \wedge X(v_2, s_2) = 1) \rightarrow \bigwedge_{l\in p(s_1,s_2)} (Y(l, e) = r(e)) \Big).$$

Therefore, the whole formula is:

$$\Phi_{PN,VN} = \alpha_s \wedge \alpha_v \wedge \beta_{server} \wedge \beta_{link}.$$

a) Abstraction: In the abstraction phase, we "compress" T into a 1-level tree T_{abs}, which has a single root r with its children $w_1, ..., w_k$. By abstracting the subtree rooted at w_i, we set the capacity of w_i to be the sum of capacities of all its children. That is, $c(w_i) = \sum_j c(s_{i,j})$. Using the encoding technique above, we build the constraint $\Phi_{T_{abs},VN}$ and solving the constraint gives us a solution for the allocation problem for the abstracted tree and the original virtual network.

b) Refinement: In the refinement phase, we need to solve for the subtree T_i with root w_i and its children $s_{i,1}, ...s_{i,m}$. By solving the abstracted tree T_{abs} in the first phase, we know the set of virtual machines that are mapped to this subtree. The virtual network VN_i that are mapped to the subtree T_i is a subgraph of the original virtual network. Suppose the set of VMs that are mapped to the subtree is V_i, then $VN_i = (V_i, E_i)$, and here $(v_1, v_2) \in E_i$ if and only if $v_1, v_2 \in V_i$ and $(v_1, v_2) \in E$. The formula Φ_{T_i,VN_i} encodes the constraint for mapping the virtual network VN_i to the subtree T_i. However, this is not sufficient to ensure that the mapping is feasible for the original virtual network. In fact, there may be a virtual link of which only one end is mapped to the subtree. To handle this situation, we need to establish a route for the server which the virtual machine is mapped to and the root switch in the subtree. That is,

$$\alpha_{v'} : \bigwedge_{\substack{(v_1,v_2)\in E, v_1\in V_i,\\ v_2\notin V_i, s_1\in T_i}} \Big(X(v_1, s_1) = 1 \rightarrow Y((s_1, w_i), e) = r(e) \Big).$$

Therefore, the formula for the refinement is

$$\Phi_{T_i,VN_i,VN} = \Phi_{T_i,VN_i} \wedge \alpha_{v'}.$$

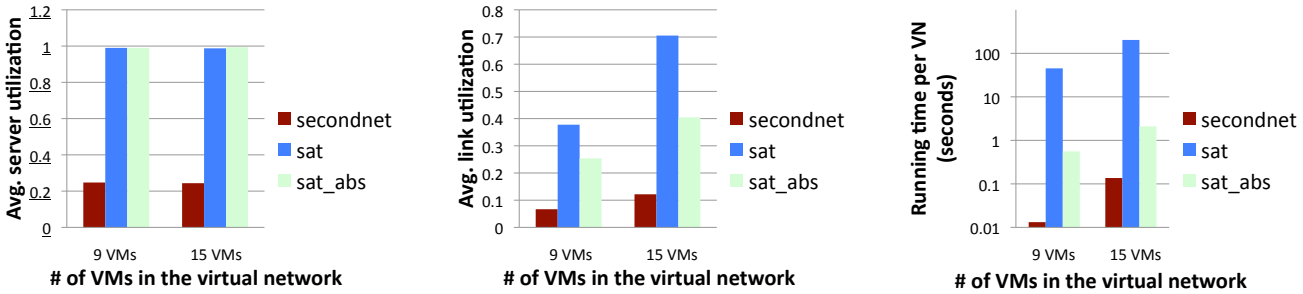
The algorithm for the abstraction and refinement approach is shown as algorithm 1. The algorithm first solves the formula of the abstracted tree (line 4), and then refines each subtree (line 5-7). To facilitate the search, if the refinement phase of some subtree fails, we stop refining the next subtree and return back to solve the first phase. To force the SMT solver to provide a different solution, we add the counter-example that makes the refinement fail. That is $\alpha_{counter} : \neg\big(\bigwedge_{v\in V_i} X(v, w_i) = 1\big)$ (line 8-11). If all the subtrees can be refined, the algorithm finds a feasible solution (line 13-17). Otherwise, the formula Φ for the abstracted tree is unsatisfied after a number of iterations, in which case, there is no feasible solution (line 19).

Algorithm 1 Abstraction&Refinement SAT solving.

```
1: build the formula Φ_{T_abs,VN} for the abstracted tree T_abs;
2: Φ = Φ_{T_abs,VN};
3: while Φ is satisfied do
4:     solve Φ;
5:     for all i do
6:         let VN_i = (V_i, E_i) be the virtual network that
       needs to be mapped to the subtree T_i;
7:         build the formula Φ_{T_i,VN_i,VN};
8:         if Φ_{T_i,VN_i,VN} is unsatisfiable then
9:             Φ = Φ ∧ ¬(⋀_{v∈V_i} X(v, w_i) = 1);
10:            break;
11:        end if
12:    end for
13:    if all subtrees can be refined then
14:        set f(v) = s if X(v, s) = 1;
15:        compute ρ using the unique routing path in T;
16:        return f and ρ;
17:    end if
18: end while
19: return no solution;
```

IV. Simulation Results

In this section, we show some empirical evaluation of our SAT solution. The topology of the physical network in our evaluation is a 2-level tree and the leaves denote the host servers and there are 2-levels of switches. There are 200 servers and each server can host 4 VMs and we set the bandwidth of the lower level link to be 20, while the higher level links' bandwidth is set to be 200. We use the topology generated by connecting 3 complete graph with 3 links as the topology of the virtual network. This topology is used to model the distributed storage systems that have 3 identical replicas, and each replica communicates with one another. We use two different size of replicas, namely 3 VMs and 5 VMs in one replica, and thus 9 VMs and 15 VMs in the virtual network respectively. In each replica, the bandwidth requirement for each virtual link ranges from 0 to 2 at random, and the bandwidth of the links connecting each replica is always 1. That is, each virtual link requires 5% to 10% bandwidth of that of the lower level links in the physical network. For comparison, we run the SAT encoding algorithm without abstraction and refinement (referred as *sat*), as well as the SAT encoding with the abstraction and refinement (referred as *sat_abs*). In addition, we compare the algorithm proposed in [1] (referred as *secondnet*). For each virtual network, we try to map as many copies of the virtual network as possible to the physical network using the allocation algorithms. We run 3 times for each virtual network, and compare the server utilization, link utilization and the running time of the 3 algorithms. All the evaluations are run on a server with quad-core 2.67GHz Intel Xeon CPU, 4GB of RAM, and we use



(a) Average Server Utilization. (b) Average Link Utilization. (c) Running Time.

Fig. 1: Simulation Results.

Z3 [5] as the SAT/SMT solver in the allocation algorithms. Figure 1a shows the *server utilization* for the 3 allocation algorithms. Server utilization measures the ratio of the number of VMs that are mapped to the physical network and the total capacity of the network. The larger the server utilization is, the more effective the allocation algorithm is. As shown in Figure 1a, both algorithms based on SAT encodings achieve 99% server utilization, while the secondnet heuristic only achieves 30% server utilization in both virtual networks with 9 VMs and 15 VMs, respectively. The reason why secondnet can only map a few virtual networks is that it only takes the requirement for servers into account, and does not consider whether there is a feasible routing path between the servers that two VMs are mapped to. Moreover, secondnet is highly sensitive to the order in which the VM connections are requested. Different orderings result in large differences in server utilization. On the other hand, the techniques based on SAT solving encode the bandwidth allocation problem completely, and they achieve high server utilization.

The *link utilization* measures the ratio between the link bandwidth utilized and the total bandwidth. When achieving the same server utilization, the allocation algorithm with lower link utilization is usually better than those with higher link utilization, because it leaves more bandwidth for future allocation. As shown in Figure 1b, secondnet only maps a few virtual networks and thus results in low link utilization. The sat algorithm achieves very high link utilization. For the virtual network with 15 VMs, the link utilization is more than 70%. Mapping the same number of virtual networks, sat_abs utilizes the link bandwidth no more than $2/3$ of that by sat. This is due to the abstraction technique. With abstraction, VMs are mapped more locally, and thus more communication happens within the subtree. Without abstraction, the two VMs on a virtual link are more likely to be mapped into two different subtrees, and thus increases the link utilization on the links between the root switch and other switches.

Figure 1c shows the running time of allocating 1 virtual network using each allocation technique. The heuristic of secondnet runs in polynomial time, and it is orders of magnitude more efficient than the other two algorithms. In particular, the sat runs for about 3 hours to map 53 virtual networks with 15 VMs, while secondnet takes no more than 2 seconds to map 13 virtual networks. It is also shown that abstraction reduces the running time significantly. To map a virtual network with 15 VMs, sat takes 200 seconds, while sat_abs only uses about 2 seconds. Let's remark that the SAT encoding approaches and existing approaches are not mutually exclusive. In practice, those approaches can be run in parallel and the first outputting feasible allocation is adopted. Moreover, SAT encoding approaches can be applied when optimizing the data center by re-locating allocated virtual networks. In this case, finding the optimal solution is more critical than running time.

V. Conclusion

Bandwidth allocation problem is the key computational problem in data center management. In this paper, we show an alternative approach that encodes the problem into SAT formulas and apply SAT/SMT solvers to solve the problem. We report simulation results showing that by using abstraction and refinement techniques, we are able to provide high quality solutions within acceptable overhead.

References

[1] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International COnference*. ACM, 2010, p. 15.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM*, 2011.

[3] Y. Zhu and M. Ammar, "Algorithms for assigning substrate network resources to virtual network components," in *Proc. IEEE INFOCOM*, vol. 2, 2006, pp. 1–12.

[4] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/1995376.1995394

[5] ——, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[6] B. Dutertre and L. De Moura, "The Yices SMT solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, p. 2, 2006.

[7] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "Sal 2," in *Computer Aided Verification*. Springer, 2004, pp. 496–500.

[8] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *Integrated formal methods*. Springer, 2004, pp. 1–20.

Computing prime implicants

David Déharbe*, Pascal Fontaine†, Daniel Le Berre‡, Bertrand Mazure‡

* UFRN, Brazil
† Inria, U. of Lorraine, France
‡ CRIL, U. of Artois, France

***Abstract*—Model checking and counter-example guided abstraction refinement are examples of applications of SAT solving requiring the production of models for satisfiable formulas. Better than giving a truth value to every variable, one can provide an implicant, i.e. a partial assignment of the variables such that every full extension is a model for the formula. An implicant is *prime* if every assignment is necessary. Since prime implicants contain no literal irrelevant for the satisfiability of the formula, they are considered as highly refined information.**

We here propose a novel algorithm that uses data structures found in modern CDCL SAT solvers to efficiently compute prime implicants starting from an existing model. The original aspects are (1) the algorithm is based on watched literals and a form of propagation of required literals, adapted to CDCL solvers (2) the algorithm works not only on clauses, but also on generalized constraints (3) for clauses and, more generally for cardinality constraints, the algorithm complexity is linear in the size of the constraints found. We implemented and evaluated the algorithm with the Sat4j library.

I. Introduction

Although SAT is a decision problem whose answer on an input formula is "satisfiable" or "unsatisfiable", it is often necessary or useful to obtain an explanation of this output, i.e. proofs of unsatisfiability for unsatisfiable formulas and models for satisfiable formulas. As a side effect of the data structures they use, modern SAT solvers output full models for satisfiable formulas, i.e. they assign a value to every variable in the input (even if the value of some variables is irrelevant). For some applications, a partial model or implicant (i.e. a partial assignment that is sufficient to satisfy all clauses) is preferred to a full assignment. Bounded model checking is one such application: an assignment corresponds to an error trace, and the smaller the assignment, the simpler it usually is to understand the flaw [1]. Using implicants instead of models is also useful when performing Boolean optimization (e.g. Pseudo Boolean Optimization or MaxSAT). Evaluating an objective function over an implicant provides a range of values (which may contain a single element) instead of a single value with a model. As such, optimization approaches based on strengthening may compute better upper bounds from implicants rather than from models. Generating partial assignments is also useful in Satisfiability Modulo Theories (see [2] for a thorough introduction) when the theory reasoner has a high complexity. Implicants are also used in the context of compilation of knowledge base, the cover of implicants being a classical way to compile a knowledge base [3]–[5].

This work has been partially supported by CAPES grant 2347-13-0, CNPq grants 308008/2012-0, 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES), Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', and ANR TUPLES.

An implicant is *prime* if none of its proper subsets is an implicant. The paper addresses the problem of efficiently deriving a prime implicant from an existing model of a satisfiable formula. A prime implicant can be derived from a model by iteratively removing the assignments that are not necessary. In this paper, we present two instances of this greedy approach. The first associates counters to constraints, yielding the algorithm sketched in [6]. This algorithm has complexity linear in the size of the constraints, but requires specialized indexing and dedicated counters as found in DPLL-based solvers. We propose a new algorithm benefiting from the lazy data structures (i.e. watched literals [7]) available in modern SAT solvers. Our approach is not only suitable for clauses but generalizes to e.g. cardinality constraints. For sets of clauses and cardinality constraints, the complexity of this algorithm is also linear, thanks to a dedicated propagation procedure on the constraints.

Related work. We focus on computing *one* prime implicant (not necessarily of minimum size) out of a given model, using the data structures used in modern SAT solvers. Algorithm 1 (Section II-C) is quickly discussed in [6] and [8], without concrete implementation or complexity study; in Section II-C we provide a concrete instantiation of it, and discuss its complexity. An algorithm embedding SAT solving techniques is proposed in [9] and motivated by experimental results. Some other techniques, e.g. [10], involve encoding the problem of finding prime implicants to linear programming. Getting minimal assignments (i.e. prime implicants) for a CNF (Conjunctive Normal Form) from a model provided by a SAT solver is discussed in [1], and several techniques are sketched. The authors of this work notably notice that literals assigned by propagation are mandatory in any prime implicant included in the model; for completeness, we restate formally this result in Section II-C. They also mention brute-force lifting, noticing it can be implemented in time quadratic in the size of the CNF formula. In the same context, the time complexity of our algorithms is linear.

A lot of research concentrates on the problem of generating one prime implicant or the set of all prime implicants for a

ISBN 978-0-9835678-3-7/13.

formula, without previous knowledge of models, e.g. [3], [8], [11]–[13]. Also, many works focus on the more complex problem of finding prime implicants of minimum size (e.g. [14] in propositional logic, and [15] in the context of SMT); the techniques presented here could be used repeatedly to find prime implicants of minimum size, but this goes beyond the scope of this paper.

Overview. Section II introduces definitions and notations. In Section II-C, we give an original formal presentation of some of the results mentioned above. Section III then presents our algorithm based on watched literals and propagation. This algorithm has been implemented in the Sat4j library [16]; experimental results are given in Section IV.

II. Basic principles

A. Definitions and notations

We assume the standard notions of propositional logic, model, propositional variable, literal and clause. A (set of) formula(s) B is a logical consequence of a (set of) formula(s) A ($A \models B$) if every model of (all elements in) A is also a model of (all elements in) B. In this paper, we use the term *constraint* for formula, implicitly understanding that a constraint c most often denotes:

- a *clause*, a disjunctive set of literals;
- a *cardinality constraint* $\sum_{\ell_i \in c} \ell_i \geq k$ where k (the degree) is an integer and each literal ℓ_i is either 0 (false) or 1 (true) — a clause can be seen as a cardinality constraint of degree 1;
- a *pseudo-Boolean constraint* $\sum_{\ell_i \in c} w_i \ell_i \geq k$, where k and each w_i are positive integers.

A set of constraints is viewed as the conjunctive combination of its elements and a *literal* as a Boolean assignment of a propositional variable. Throughout this paper, a *set of literals* cannot contain two opposite literals, so that sets of literals essentially are partial mappings from the lexicon of propositional variables to the Boolean values. In the following we identify a model for a (set of) formula(s) with the set of all the literals it satisfies.

A set of literals M is an *implicant* for a set of constraints $\mathcal{C}$ if, for every constraint $c \in \mathcal{C}$, $M \models c$. An implicant M of $\mathcal{C}$ is a *prime implicant* if, for every proper subset M' of M, M' is not an implicant of $\mathcal{C}$. Assuming $M \models c$ and $\ell \in M$, we say ℓ is a *required literal* in M for constraint c, and write $Req(M, \ell, c)$, when $M \setminus \{\ell\} \not\models c$. In particular, for a clause c such that $M \models c$, we have $Req(M, \ell, c)$ iff $M \cap c = \{\ell\}$. A *required literal* ℓ for M and a set of constraints $\mathcal{C}$, denoted $Req(M, \ell, \mathcal{C})$, is such that there exists a constraint $c \in \mathcal{C}$ with $Req(M, \ell, c)$.

B. Elements of SAT solving

Modern CDCL-based SAT solvers assume their input is given as a set of clauses, but the techniques described here may be generalized to handle cardinality and pseudo-Boolean constraints. To decide if a set of clauses is satisfiable, a solver must find a variable assignment that satisfies all clauses. Three key aspects of this search are decision, propagation and learning. *Decision* consists in setting an unassigned variable to a Boolean value. A variable assignment is *propagated* if it is enforced by the previous assignments, i.e. this happens when all but one literal in a clause have been assigned to false. Then this last literal must be true for the set of clauses to be satisfiable. It may happen that propagation implies a conflicting assignment. In that case, a new clause (the conflict) is *learnt*, being recorded as a new constraint. Then backtracking and further propagation occur. If propagation terminates without conflict, either all variables are assigned and the set is satisfiable, or a new decision occurs. On an unsatisfiable set of constraints, the algorithm will eventually reach a conflicting assignment with no decided variable.

In practice, the computation cost is dominated by propagation. A naïve algorithm could be: whenever a variable is assigned a value, all clauses containing the literal set to false are checked for unsatisfiability or new propagations. The *watched literals* technique is a heuristic that effectively reduces that cost in practice. In the case of clauses, it is based on the observation that a clause needs to be inspected only when all but one literal are assigned to false. So, for each clause, only two of its literals are watched, and the clause is inspected only when one of the two watched literals is assigned to false. This technique generalizes to cardinality constraints, by watching at most $k+1$ literals, for a constraint of degree k.

C. Greedy computation of prime implicants from models

Consider a model M for a set of constraints $\mathcal{C}$. Most often, the model M is computed with a solver using propagation; knowing which literals in M are propagated, and which are not, is highly valuable information for computing prime implicants out of M. Indeed, the following simple lemma allows to directly identify elements in M that have to be in every prime implicant included in M.

Lemma 1: Assume 1) M is an implicant for a set of formulas $\mathcal{C}$, 2) c is a logical consequence of $\mathcal{C}$, 3) and $M \setminus \{\ell\} \not\models c$. Then $M \setminus \{\ell\}$ is not an implicant of $\mathcal{C}$. In other words, the literal ℓ belongs to every prime implicant included in M.

Proof. If c is a logical consequence of $\mathcal{C}$, then every implicant of $\mathcal{C}$ is an implicant of c. As $M \setminus \{\ell\}$ is not an implicant of c, $M \setminus \{\ell\}$ is not an implicant of $\mathcal{C}$. □

In the context of CDCL solvers, the above trivial lemma has an interesting corollary. Assume $\ell \in M$ is propagated, i.e. there exists a constraint c in $\mathcal{C}$ or learnt from $\mathcal{C}$ — in both cases, c is a logical consequence of $\mathcal{C}$ — and a subset $M' \subseteq M \setminus \{\ell\}$ such that $M', c \models \ell$. Then M, $\mathcal{C}$, c and ℓ fulfill the requirements of the lemma: ℓ is mandatory in every implicant included in M. Only decision literals may possibly be removed from M to obtain a stronger implicant.

The abstract Algorithm 1 computes a prime implicant for a set of constraints $\mathcal{C}$, starting from a model M_0 of $\mathcal{C}$ and a subset Π_0 of the literals known to be in a prime implicant (e.g., the empty set, or the set of all propagated literals in the CDCL solver that produced M_0). Variable M is an implicant for $\mathcal{C}$ of decreasing size, and Π is an increasing subset of a prime implicant included in M. The algorithm checks each

literal ℓ in $M \setminus \Pi$ and greedily adds it to Π if it is required or removes it from M otherwise. There may be several different prime implicants included in M_0; the successive choices of ℓ in line 3 determine which of those prime implicants is returned by the algorithm.

Algorithm 1 Abstract computation of prime implicants

```
1: procedure PRIME(C, M_0, Π_0)
2:     M, Π ← M_0, Π_0
3:     while ℓ ∈ M \ Π do
4:         if Req(M, ℓ, C) then Π ← Π ∪ {ℓ}
5:         else M ← M \ {ℓ}
6:     return Π
```

The algorithm can be refined in a practical and efficient algorithm. Remember that checking if $Req(M, \ell, \mathcal{C})$ is true comes to check if $Req(M, \ell, c)$ is true for some constraint in $c \in \mathcal{C}$. It is thus useful, in order not to check every constraint in $\mathcal{C}$, to have an index $W(\ell)$ that gives the set of constraints containing ℓ. This index can be built efficiently, though it requires to read the entire set of constraints.

Algorithm 2 was sketched in [6] and is specialized for sets of clauses; it can be extended easily (at the expense of heavier notations) to cardinality constraints while preserving the linear complexity. It can also be extended to arbitrary constraints, but requires to define the concrete code for $Req(M, \ell, c)$ for an arbitrary constraint c. If c is a clause, $Req(M, \ell, c)$ is true if and only if $M \cap c = \{\ell\}$. Such a test can be done efficiently using counters for the true literals in every clause c; in Alg. 2, line 10, $\exists c \in W(\ell)$. $\mathrm{N}[c] = 1$ stands for a loop on $W(\ell)$ that stops returning true if $\mathrm{N}[c] = 1$ for some c, and returns false otherwise. For every clause c, $\mathrm{N}[c]$ has to be initialized to $|M_0 \cap c|$. The counters in N have to be updated each time a literal is removed from M (line 13).

Algorithm 2 Prime implicants for CNFs.

```
1: procedure PRIME(C, M_0, Π_0)
2:     M, Π ← M_0, Π_0
3:     for all ℓ ∈ M do W(ℓ) ← ∅
4:     for all c ∈ C do
5:         N[c] ← 0
6:         for all ℓ ∈ c do W(ℓ) ← W(ℓ) ∪ {c}
7:     for all ℓ ∈ M do
8:         for all c ∈ W(ℓ) do N[c] ← N[c] + 1
9:     for all ℓ ∈ M \ Π do
10:        if ∃c ∈ W(ℓ) . N[c] = 1 then
11:            Π ← Π ∪ {ℓ}
12:        else
13:            for all c ∈ W(ℓ) do N[c] ← N[c] − 1
14:            M ← M \ {ℓ}
15:    return Π
```

Theorem 1: Given a satisfiable set of clauses $\mathcal{C}$, a model M_0, and a set Π_0 of literals mandatory in all prime implicants included in M_0, Algorithm 2 returns a prime implicant for $\mathcal{C}$. It runs in time $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$.

Proof. If the set of literals given as argument of the function is a model, then the returned set of literals is also a (partial) model for $\mathcal{C}$. Indeed, a literal ℓ is removed from the model if and only if all clauses are still satisfied when ℓ is removed.

Furthermore, the returned partial model M is minimal. Assume $M \setminus \{\ell\}$ is also a partial model for $\mathcal{C}$. If ℓ has not been removed, either there exists a clause $c \in \mathcal{C}$ such that ℓ is the sole true literal, or ℓ was initially in Π_0. In the first case, $M \setminus \{\ell\}$ cannot be a partial model for c and hence for $\mathcal{C}$. The second case would contradict the theorem hypothesis on Π_0.

Assume that, for each clause c, the counter $\mathrm{N}[c]$ can be read and modified in constant time. Assume also that, for each ℓ, the indexing $W(\ell)$ of clauses containing literal ℓ is such that 1) it can be emptied in constant time, 2) an element can be added in constant time, 3) all its elements can be iteratively read in cumulative linear time. We also suppose that iterating on $\mathcal{C}$, M and $M \setminus \Pi$ has a cumulative cost which is respectively $\mathcal{O}(|\mathcal{C}|)$, $\mathcal{O}(|M|)$, and $\mathcal{O}(|M|)$.

Under the above assumptions, Algorithm 2 is linear with respect to the size of the clause set $\sum_{c \in \mathcal{C}} |c|$. We consider that every literal is present in at least one clause so that $\mathcal{O}(\sum_{c \in \mathcal{C}} |c| + |M|) = \mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$. Line 3 is $\mathcal{O}(|M|)$. Lines 4–6 involve inspecting each clause and each literal in the clause, and execute a constant time operation (at line 6) for each of those literals. This block is thus $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$. Lines 7–8 involve inspecting each clause c at most $|c|$ times, and is thus also $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$. In the last loop at lines 9-14, each clause c from $\mathcal{C}$ is again examined at most $2 \times |c|$ times. Overall, all four loops are $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$. □

Algorithm 2 has linear complexity, but requires to build an index of constraints by literals and counters. Also, a constraint is examined once for every of its literals satisfied in the model (not unlike what happens in SAT solving with counters instead of watched literals). Rather than preventing a counter from decreasing to 0 (which, for SAT solving, would correspond to a conflict), it is more reasonable to directly put in Π the last satisfied literal of a clause as soon as the counter reaches one (i.e. using some kind of propagation). This motivates the version presented in the next section, using watched literals, instead of indexes and counters.

III. Computing prime implicants by propagation

It can be argued that the above algorithm uses late detection of literals to add in the prime implicant. Indeed, a literal ℓ is iteratively selected, and $Req(M, \ell, c)$ is checked for every constraint c containing ℓ. Another possibility is to use early detection of literals for addition to Π, similarly to Boolean constraint propagation in SAT solvers. This yields the algorithms described in this section.

A. An abstract version

Algorithm 3 is the early detection equivalent of abstract Algorithm 1: it computes a prime implicant out of an implicant

M_0 for a set of constraints $\mathcal{C}$, and any subset Π_0 of the required literals in M_0. Variable M, initialized to M_0, is an implicant for $\mathcal{C}$ of strictly decreasing size, and Π is an increasing subset of a prime implicant included in M. The larger Π_0 is, the faster the convergence[1]; also it is optional as the algorithm is sound if it is empty. We introduce it for future specializations.

Algorithm 3 Abstract propagation-based algorithm

```
1: procedure PRIME(C, M_0, Π_0)
2:     M, Π ← M_0, Π_0
3:     Π ← Π ∪ IMPLIED(C, M)
4:     while ℓ ∈ M \ Π do
5:         M ← M \ {ℓ}
6:         Π ← Π ∪ IMPLIED(C, M)
7:     return Π
```

Algorithm 3 first identifies and adds to Π the required literals of M (l. 3). Repeatedly one of the remaining literals in $M \setminus \Pi$ is removed (l. 4) until $M \setminus \Pi$ is empty. This may trigger other remaining literals to be added to Π (l. 6). The call $\textsc{Implied}(\mathcal{C}, M)$ yields a subset of M such that

$$\textsc{Implied}(\mathcal{C}, M) \setminus \Pi = \big\{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\big\} \setminus \Pi,$$

i.e. $\textsc{Implied}(\mathcal{C}, M)$ returns the set of literals in M that should be added to Π because, for each of these literals, a constraint requires this literal to be true. Note that, in contrast to Algorithm 1, the literal chosen in l. 4 is removed from the prime implicant without further test. Lines 3 and 6 establish the property that no literal in $M \setminus \Pi$ is required.

Proposition 1: Given a set of constraints $\mathcal{C}$, an implicant M_0, and Π_0, a subset of $\{\ell \mid \mathit{Req}(M_0, \ell, \mathcal{C})\}$, Algorithm 3 terminates and returns a prime implicant of $\mathcal{C}$ included in M_0.

Proof. The loop in Algorithm 3 satisfies the following invariants:

I_1: $\Pi = \{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$;
I_2: $\Pi \subseteq M \subseteq M_0$;
I_3: M is an implicant.

Invariant I_1 is verified at the start of the loop as a consequence of line 3 (assuming the pre-condition $\Pi_0 \subseteq \{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$ for the call to PRIME) and is preserved thanks to line 6. I_2 is trivial, and I_3 is verified at the start of the loop as a consequence of line 2. It is preserved since ℓ at lines 4 and 5 is not in Π, thus is not required: $\forall c \,.\, M \setminus \{\ell\} \models c$. The new value of M is again an implicant for all c.

The loop variant $|M \setminus \Pi|$ is a strictly decreasing sequence of natural numbers; the loop terminates when $M \subseteq \Pi$, i.e. when $M = \Pi$ (thanks to invariant I_2) and $\Pi \subseteq M_0$. From invariant I_3, Π is an implicant and from invariant I_1, this implicant is prime. This establishes the property.

The above proof is suitable for any type of Boolean constraints. For the special case of a clause $c \in \mathcal{C}$, notice that, as a direct consequence of the loop invariant, $c \cap \Pi \neq \emptyset \vee |c \cap M| \geq 2$. □

[1]Technically, a SAT solver should assign Π_0 to the set of literals assigned by unit propagation while establishing $M_0 \models \mathcal{C}$.

There may exist several prime implicants in M_0. The one produced by Algorithm 3 depends only on the successive choices of ℓ in line 4. Any prime implicant subset of M_0 may be produced, given the right sequence of chosen literals. A prime implicant produced by Algorithm 1 or Algorithm 2 is obtained by Algorithm 3 by picking literals in the same sequence and dropping literals that are already in prime.

B. Implementation with watched literals

A concrete implementation of the above abstract algorithm would best use the data structures implemented in state-of-the-art SAT solvers. This is the approach of Algorithm 4: in addition to the model M_0, it reuses the watched literals relation at the ending state of the SAT solver. We consider a general notion of watched literals as a relation W between literals and constraints such that, for every literal ℓ, $W(\ell)$ is a (sub)set of constraints containing ℓ. We now require Π_0 to initially contain all the literals that are directly entailed by one constraint in $\mathcal{C}$.[2] Since such literals are included in $\{\ell \mid \mathit{Req}(M, \ell, \mathcal{C})\}$ the precondition for Algorithm 3 is verified.

Algorithm 4 Prime implicants using watched literals

```
 1: procedure PRIME(C, M_0, Π_0, W)
 2:     M, Π ← M_0, Π_0
 3:     IMPLIED_{W,0}(C, M, Π, W)
 4:     while ℓ ∈ M \ Π do
 5:         M ← M \ {ℓ}
 6:         IMPLIED_W(C, M, ℓ, Π, W)
 7:     return Π

 8: procedure IMPLIED_{W,0}(C, M, ref Π, ref W)
 9:     for all ℓ ∈ M \ Π do
10:         IMPLIED_W(C, M, ¬ℓ, Π, W)

11: procedure IMPLIED_W(C, M, ℓ, ref Π, ref W)
12:     W_ℓ ← W(ℓ)
13:     for all c ∈ W_ℓ do
14:         HDL_CONSTR(c, M, ℓ, Π, W)
```

The data in Algorithm 4 includes the variables of Algorithm 3, namely M and Π, and the watched literals relation W. The inherent property of the watched literals for a constraint c, i.e. $W^{-1}(c)$, is that, as long as all watched literals remain either undefined or true, nothing can be deduced from c in the current assignment. In our context Π plays a role similar to the current partial assignment in the SAT solver. Let us define, for a constraint c, the set of literals $S(c) = \Pi \cup W^{-1}(c)$. Formally, W is always such that:

$W_1(c)$: $\forall \ell \in W^{-1}(c) \setminus \Pi \,.\, \neg \mathit{Req}(S(c), \ell, c)$
$W_2(c)$: $S(c) \cap M \models c$

Both properties should be satisfied by the inputs given to our algorithm. Observe that: 1) if $\Pi \models c$, then $W_1(c)$ is true; 2)

[2]In particular, Π_0 should contain all literals in unit clauses.

when a literal ℓ is removed from M, $W_1(c)$ is not affected; and 3) if ℓ furthermore satisfies $\neg Req(M, \ell, c)$, then $W_2(c)$ is also preserved. The algorithm first establishes an additional loop invariant (line 3):

$W_3(c)$: $S(c) \subseteq M$

As in Algorithm 3, the main loop repeatedly removes one (unrequired) literal ℓ from $M \setminus \Pi$ (line 5), possibly augmenting Π with new literals, and repairs the invariant properties for the watched literals (line 6). Function HDL_CONSTR (Algorithm 5) reestablishes these properties for each c affected by the removal of ℓ from M. Its definition is left general enough so that it can be specialized for different classes of constraints and watched literals strategies. This greedy approach is similar to Boolean propagation in SAT solving, Π emulating the assignment whereas M restricts the choice for watched literals and possible propagations.

Algorithm 5 HDL_CONSTR for arbitrary constraints

```
1: procedure HDL_CONSTR(c, M, ℓ, ref Π, ref W)
2:     Π ← Π ∪ {ℓ' ∈ W^{-1}(c) | Req(M, ℓ', c)}
3:     if Π ⊭ c then
4:         Choose W' such that
5:             W' ⊆ (W^{-1}(c) ∪ M) \ {ℓ}
6:             (Π ∪ W') ∩ M ⊨ c
7:             ∀ℓ' ∈ W' \ Π . ¬Req(W' ∪ Π, ℓ', c)
8:         in W^{-1}(c) ← W'
```

Proposition 2: Given a set of constraints $\mathcal{C}$, an implicant M_0, a set of literals Π_0, and a relation W between literals and constraints in $\mathcal{C}$ such that:

- $\{\ell \mid \exists c \in \mathcal{C} \,.\, c \models \ell\} \subseteq \Pi_0 \subseteq \{\ell \mid Req(M_0, \ell, \mathcal{C})\}$,
- $\forall c \in \mathcal{C} \,.\, W_1(c) \wedge W_2(c)$

then Algorithm 4 terminates and returns a prime implicant of $\mathcal{C}$ contained in M_0.

Proof. The proof is similar to that of Algorithm 3, the same invariants being satisfied: we prove that lines 3 and 6 establish these invariants through the successive calls to function HDL_CONSTR (Algorithm 5).

First, consider the call in line 3. Before the call, invariants I_2 and I_3 are satisfied, as well as $W_1(c)$ and $W_2(c)$ for each constraint c, and $\Pi \subseteq \{\ell \mid Req(M_0, \ell, \mathcal{C})\}$, thanks to the preconditions of PRIME. The call establishes $W_3(c)$ for every constraint c, and at the same time, introduces literals in Π so that $\{\ell \mid Req(M_0, \ell, \mathcal{C})\} \subseteq \Pi$. This is a direct consequence of line 2 in Algorithm 5, the other lines ensuring that $W_1(c)$, and $W_2(c)$ remain preserved even if $\bar{\ell}$ is removed. When all the negations of literals in M have been removed from the watched literals by the successive calls, $W_3(c)$ is also satisfied for each c. Every element ℓ added in Π can be related to a constraint c such that $Req(M, \ell, c)$.

Now consider the call in l. 6. The invariants are satisfied, if it were not for the absence of ℓ in M. Again, for each constraint c, the successive calls to HDL_CONSTR repair the invariant $W_3(c)$ while preserving $W_1(c)$ and $W_2(c)$. This may insert new literals in Π if they are required by c. □

Algorithm 6 HDL_CONSTR for clause or cardinality constraints

```
1: procedure HDL_CONSTR(c, M, ℓ, ref Π, ref W)
2:     if ∃ℓ' ∈ c ∩ M . ℓ' ∉ W^{-1}(c) then
3:         W ← (W ∪ {ℓ' ↦ c}) \ {ℓ ↦ c}
4:     else Π ← Π ∪ (W^{-1}(c) \ {ℓ})
```

Function HDL_CONSTR in Algorithm 5 is generic and may be refined to handle specific classes of constraints. One such concrete implementation is given for clauses and, more generally, for cardinality constraints in Algorithm 6. Assuming $|W^{-1}(c)| \geq 2$, $c \in W(\ell)$ and $\ell \notin M$, either there exists another literal ℓ' that may be watched by c, in which case W is updated with the new association, or there is no such literal, and the literals in $W^{-1}(c)$ must be in the prime implicant and are inserted into Π. In the special case of clauses, then there is only one such literal.

Proposition 3: When $\mathcal{C}$ is a set of clauses and HDL_CONSTR is specified as in Algorithm 6, Algorithm 4 runs in time $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$.

Proof. IMPLIED$_W(\mathcal{C}, M, \ell, \Pi, W)$ has cumulated complexity in $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$. To achieve this rate, one has to ensure that, for every clause c, the cumulative time for the calls to HDL_CONSTR with c is $\mathcal{O}(|c|)$. This can simply be done by storing clauses as arrays of literals indexed from 1 to $|c|$, using a pointer initialized to 1 in this array, and looking for the suitable literal from this pointer on (and updating its value). The successive calls to HDL_CONSTR on clause c would then resume their search from the previous position. Each literal in each clause would therefore be processed at most once.

For every literal ℓ in M, there is one call to IMPLIED$_W(\mathcal{C}, M, \bar{\ell}, \Pi, W)$ in function IMPLIED$_{W,0}$. Every clause in $\mathcal{C}$ is satisfied by at least one of its watched literals. If a clause appears in $W(\bar{\ell})$, its other watched literal is thus in M, and the watched $\bar{\ell}$ will be replaced by another watched literal in M (or the clause stays in $W(\bar{\ell})$ and its other watched literal is added to Π). So every clause will be examined at most once for the whole run of IMPLIED$_{W,0}$. Assuming the search for another watched literal in line 2 of Algorithm 6 remains linear with respect to the size of the clause, IMPLIED$_{W,0}$ runs in time $\mathcal{O}(\sum_{c \in \mathcal{C}} |c|)$.

IMPLIED$_W$ is called at most once for each literal ℓ in M_0 on line 6 in Algorithm 4. If the watch relation for clause c is modified (on line 3 in Algorithm 6), c will never occur again in $W(\ell)$, since ℓ is removed forever from M. As a consequence, every clause c will be considered at most $|c|$ times by the successive calls of Algorithm 6. In these calls, the cumulated searches for a new watched literal (condition on line 2 in Alg. 6) accounts for a factor linear in the size of c. □

Prop. 3 may be generalized to any class of constraints and watched literals strategy where the cumulated time of HDL_CONSTR(c) has a complexity linear in the size of c. This holds for cardinality constraints, as the watched literals

strategy may also be employed.

IV. Experimental evaluation

A classical implementation of Algorithm 1 with quadratic complexity in the size of $M_0 \setminus \Pi_0$ has already been available in Sat4j for several years. In practice, this implementation performed well on many SAT benchmarks because a vast majority of the literals of the model found by the SAT solver are implied by unit propagation, so $M_0 \setminus \Pi_0$ was initially much smaller than M_0 (Π_0 containing all propagated literals initially). There are however classes of problems for which this is not true.

Sat4j MaxSAT uses selector variables to translate MaxSAT problems into Pseudo-Boolean Optimization problems [16]. In that context, counting the number of satisfied selector variables provides an upper bound on the minimum number of constraints that must be falsified. However, despite a strategy to always branch first on falsified selector variables, some selector variables may be satisfied even if the original constraint is satisfied. To improve the bounds, two solutions exist: using an encoding enforcing that the selector variable can only be satisfied if the original constraint is falsified, or counting the selector variables on a prime implicant. The former solution adds many binary clauses to the original CNF (as many binary clauses as literals in the original formula) and is inefficient in practice.

We present here some experimental results of the proposed algorithm on a specific set of benchmarks from the MaxSAT 2010 evaluation. The previous version of Sat4j could not compute prime implicants for industrial MaxSAT benchmarks from *circuit debugging* with millions of variables and clauses [17]. We used those benchmarks to compare the proposed algorithm based on watched literals against one based on counters, both of linear complexity.

Since we use prime implicants to improve the upper bounds computed by our MaxSAT solver, prime implicants have to be computed on a set of clauses plus one cardinality or pseudo-boolean constraint representing the bound of the objective function. On the following, we present the time required to compute the first prime implicant of each benchmark, thus on clauses only.

Algorithm 4 (for clauses) and Algorithm 6 (for clauses, cardinality and pseudo-Boolean constraints) have been implemented in the Sat4j library. As described in the previous section, the implementation includes a propagation procedure similar to the classical unit propagation scheme found in CDCL solvers with two key differences: i) the propagation always eventually finds a satisfied literal and ii) the number of steps to update the watched literals is reduced by storing the last position in the search between each call to the propagation procedure. Note that for clauses, where only two literals are watched, a constraint with n literals is traversed at most n times if there is no bookkeeping, and it may be a good tradeoff to avoid storing that information for short clauses to save memory. For larger clauses, or cardinality constraints, bookkeeping the state of the search as proposed for Algorithm 6 is crucial: on some examples, the time spent to compute a prime implicant was dramatically reduced (e.g. from 240 seconds to less than one second) by such a simple implementation detail, that guarantees the linearity of the algorithm. For pseudo-Boolean constraints, we use a counter based implementation and extra care is required to update the state during backtracking and to handle the literals that do not belong to the implicant. Finally, learned clauses are ignored for the propagation. The implementation details can be found in the source code of Sat4j.

The results are summarized in Table I. The Sean Safarpour benchmark set contains 52 benchmarks. Sat4j is able to load 36, running out of memory for the others (when given 2GB of memory). For those 36 benchmarks, we give the number of variables (including the selector variables, one per clause), the number of clauses, the total number of literals in the formula (the cumulated size of the clauses), the number of literals implied by unit propagation in the model (#implied), and the time taken respectively by the counter *vs.* watched literals approaches to compute a prime implicant from the first model found by the MaxSAT solver. We also give the median values on those 36 benchmarks. The proposed algorithm was able to compute prime implicants for all benchmarks within a second, while the counter based approach missed one (due to memory out) and lead in some cases to much greater runtimes (up to one order of magnitude). Those results illustrate the advantage of reusing the solver data structures to minimize memory requirements and the advantage in practice of using those lazy data structure for computing prime implicants.

V. Conclusion

We propose and discuss an algorithm to compute prime implicants in time linear in the size of the input formula designed for easy integration in modern SAT solvers. This algorithm is based on lazy data structures such as watched literals [7]. The efficiency of the algorithm is maintained for other kinds of constraints as long as some data structure ensures the constraint will be traversed at most once during the successive calls to the propagation procedure. This applies to both clauses and cardinality constraints. The same algorithm can also be applied to other kind of constraints, but linear complexity may be lost.

We implemented the algorithm for clauses, cardinality and pseudo-Boolean constraints in the Sat4j platform. On a class of problems with millions of variables, we compare a counter based algorithm against our watched literal algorithm. While both algorithms are linear, our algorithm computed all prime implicants in less than a second, which was not the case for the other algorithm. These results show that applying the proposed algorithm to compute prime implicants instead of models has a negligible overhead.

Good prime implicant computation procedures are useful for many applications. In particular, we investigate prime implicants for Boolean optimization by strengthening, as the value of the objective function computed on a prime implicant

TABLE I
EXPERIMENTAL COMPARISON OF THE PRIME IMPLICANT ALGORITHMS ON SELECTED SEAN SAFARPOUR BENCHMARKS (2GB MEMORY).

#vars (M)	#cla (M)	#literals (M)	#implied (M)	Alg. 2 (s)	Alg. 4 (s)
2.3	1.7	4.0	0.5	4.842	**0.736**
1.3	0.9	2.2	0.4	**0.347**	0.377
1.5	1.1	2.7	0.4	2.860	**0.495**
2.6	1.8	4.4	0.6	MO	**3.463**
1.5	1.0	2.5	0.3	0.541	**0.380**
0.7	0.5	1.3	0.2	**0.210**	0.230
0.7	0.5	1.3	0.2	**0.212**	0.237
1.0	0.7	1.8	0.3	0.729	**0.364**
0.9	0.7	1.8	0.2	**0.225**	0.252
1.0	0.7	1.9	0.2	0.559	**0.283**
1.0	0.7	1.9	0.2	0.552	**0.283**
1.0	0.8	2.1	0.2	0.578	**0.301**
0.2	0.16	0.4	0.04	0.154	**0.120**
0.5	0.4	1.1	0.1	0.552	**0.221**
0.2	0.9	2.4	0.25	**0.280**	0.353
2.0	1.5	3.9	0.5	4.191	**0.486**
1.6	1.2	2.9	0.4	3.956	**0.377**
1.0	0.8	2.1	0.2	0.638	**0.284**
1.8	1.0	2.8	0.3	4.008	**0.354**
2.0	1.6	4.5	0.4	2.567	**0.486**
1.1	0.9	2.6	0.2	0.326	**0.304**
1.1	0.9	2.6	0.2	0.333	**0.289**
1.1	0.9	2.6	0.2	**0.319**	0.330
1.1	0.9	2.6	0.2	**0.343**	0.684
2.0	1.6	4.6	0.4	2.493	**0.493**
0.8	0.7	1.9	0.1	**0.232**	0.269
1.2	0.9	2.5	0.2	0.621	**0.348**
0.2	0.1	0.3	0.04	0.152	**0.102**
0.2	0.1	0.3	0.04	0.154	**0.077**
2.2	1.7	4.8	0.4	9.225	**0.510**
2.2	1.7	4.8	0.4	8.946	**0.490**
2.2	1.7	4.8	0.4	6.086	**0.556**
1.5	1.2	3.4	0.3	4.250	**0.366**
1.5	1.2	3.4	0.3	4.172	**0.370**
1.0	0.8	1.9	0.3	0.643	**0.285**
1.0	0.8	1.9	0.3	0.645	**0.273**
Median					
1.168	0.930	-	0.268	0.578	0.301

yields a better upper bound than the value obtained with a model.

REFERENCES

[1] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *TACAS*, 2004, vol. 2988 of *LNCS*.

[2] C. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. 2009.

[3] R. Schrag, "Compilation for critically constrained knowledge bases," in *AAAI*, 1996, pp. 510–515.

[4] Y. Boufkhad, É. Grégoire, P. Marquis, B. Mazure, and L. Saïs, "Tractable cover compilations," in *IJCAI*, 1997.

[5] A. Darwiche and P. Marquis, "A knowledge compilation map," *J. Artificial Intelligence Research*, vol. 17, 2002.

[6] T. Castell, "Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure," in *ICTAI*, 1996.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *DAC*, 2001.

[8] L. Palopoli, F. Pirri, and C. Pizzuti, "Algorithms for selective enumeration of prime implicants," *Artif. Intell.*, vol. 111, no. 1-2, 1999.

[9] S. Shen, Y. Qin, and S. Li, "Minimizing counterexample with unit core extraction and incremental SAT," in *VMCAI*, R. Cousot, Ed., 2005, vol. 3385 of *LNCS*.

[10] C. Pizzuti, "Computing prime implicants by integer programming," in *ICTAI*, 1996.

[11] A. Kean and G. Tsiknis, "An incremental method for generating prime implicants/implicates," *J. Symbolic Computation*, vol. 9, no. 2, 1990.

[12] N. V. Murray and E. Rosenthal, "Linear response time for implicate and implicant queries," *Knowl. Inf. Syst.*, vol. 22, no. 3, 2010.

[13] A. Ramesh, G. Becker, and N. V. Murray, "CNF and DNF considered harmful for computing prime implicants/implicates," *J. Automated Reasoning*, vol. 18, no. 3, 1997.

[14] V. Manquinho, P. Flores, J. P. Marques Silva, and A. Oliveira, "Prime implicant computation using satisfiability algorithms," in *ICTAI*, 1997.

[15] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken, "Minimum satisfying assignments for SMT," in *Computer Aided Verification (CAV)*, 2012, vol. 7358 of *LNCS*.

[16] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, 2010.

[17] S. Safarpour, H. Mangassarian, A. Veneris, M. Liffiton, and K. Sakallah, "Improved design debugging using maximum satisfiability," in *FMCAD*, 2007.

A Circuit Approach to LTL Model Checking

Koen Claessen
koen@chalmers.se

Department of CSE, Chalmers University of Technology
Gothenburg, Sweden.

Niklas Een, Baruch Sterin
{een,sterin}@eecs.berkeley.edu

Department of EECS, University of California, Berkeley, USA.

***Abstract*—This paper presents a method for translating formulas written in assertion languages such as LTL into a monitor circuit suitable for model checking. Unlike the conventional approach, no automata is generated for the property, but instead the monitor is built directly from the property formula through a recursive traversal. This method was first introduced by Pnueli et. al. under the name of *Temporal Testers*. In this paper, we show the practicality of temporal testers through experimental evaluation, as well as offer a self-contained exposition for how to construct them in manner that meets the requirements of industrial model checking tools. These tools tend to operate on logic circuits with sequential elements, rather than transition relations, which means we only need to consider so called *positive testers* with no *future references*. This restriction both simplifies the presentation and allows for more efficient monitors to be generated. In the final part of the paper, we suggest several possible optimizations that can improve the quality of the monitors, and conclude with experimental data.**

I. At a glance

Consider the LTL formula:

$$\textit{for-all-paths: } \mathbf{G}\neg a \ \vee \ \mathbf{XF}\neg b$$

A witness to its negation satisfies:

$$\textit{there-exists-a-path: } \mathbf{F}a \ \wedge \ \mathbf{XG}b$$

If no such witness exists, the original formula holds. Construct the following equisatisfiable formula by introducing a variable for each subformula, including the full formula:

$$\begin{array}{ll} & z_0 \\ \wedge & \mathbf{G}(z_0 \ \leftrightarrow \ z_1 \wedge z_2) \\ \wedge & \mathbf{G}(z_1 \ \leftrightarrow \ \mathbf{F}a) \\ \wedge & \mathbf{G}(z_2 \ \leftrightarrow \ \mathbf{X}z_3) \\ \wedge & \mathbf{G}(z_3 \ \leftrightarrow \ \mathbf{G}b) \end{array} \qquad (1)$$

Since the specification is in *negated normal form* and all the operators are monotonic, bi-implications can be replaced by simple implications:

$$\begin{array}{ll} & z_0 \\ \wedge & \mathbf{G}(z_0 \ \rightarrow \ z_1 \wedge z_2) \\ \wedge & \mathbf{G}(z_1 \ \rightarrow \ \mathbf{F}a) \\ \wedge & \mathbf{G}(z_2 \ \rightarrow \ \mathbf{X}z_3) \\ \wedge & \mathbf{G}(z_3 \ \rightarrow \ \mathbf{G}b) \end{array} \qquad (2)$$

Two types of properties commonly supported by modern model checking tools are:

- **Safety.** A counterexample is a *finite* path to a *bad* state.
- **Liveness.** A counterexample is an *infinite* path where a set of signals $f_1, f_2, \ldots, f_k$ are each true infinitely often.

In the following sections, it is shown how the conjuncts in (2) can each be translated into a small monitor circuit together with a liveness property, yielding a new model that can be verified by existing model checking tools. Furthermore, it is shown how the safety fragment of a temporal formula can be checked more efficiently by producing a safety property for that part.

Example. The conjunct $\mathbf{G}(z_1 \ \rightarrow \ \mathbf{F}a)$ of (2) is translated into a circuit that outputs TRUE as long as $z_1 = 0$, then when $z_1 = 1$, it starts waiting for $a = 1$, outputting FALSE in the meanwhile. When $a = 1$ arrives, the circuit goes back to waiting for $z_1 = 1$, while again outputting TRUE. This output signal needs to hold infinitely often for a witness to the formula, and is thus added as a liveness property.

II. Introduction

A. Automata-theoretic approach

Vardi and Wolper [18] introduced the automata-theoretic approach to verification. Given a formula ϕ and a machine M, finding whether $M \models \phi$ is done by creating an automaton $A_{\neg\phi}$ that accepts the traces that violate ϕ, and then checking whether $M \times A_{\neg\phi}$ is empty. In this paper we discuss generating circuits representing finite automata for detecting finite traces and simple Büchi automata for liveness properties.

B. Related work

Vardi and Wolper [18] showed that every LTL formula can be translated to a Büchi automaton that accepts the same language. There are now many approaches to perform that translation. In this section we review the most common ones.

The first set of approaches use direct construction of a Büchi automaton. These methods tend to be complicated, and may generate exponentially large automata.

The second set [17] translates the LTL formula into an alternating automaton, which is then translated into a Büchi automaton. The main advantage is the simplicity of the resulting alternating automaton, whose size is linear in the size of the formula. The resulting Büchi automaton has an exponential number of states in the size of the formula, but the size of the symbolic description is linear. This approach is also compositional; the alternating automaton for a formula is obtained from the alternating automata for its subformulas.

ISBN 978-0-9835678-3-7/13.

An often overlooked problem with this approach is that a good understanding of this flow, and especially of alternation and its removal [5], is a non-trivial intellectual undertaking. In an industrial environment, a simpler approach, especially if it has few disadvantages, is to be preferred.

Kesten et. al. in [10], and later Pnueli and Zaks in [13], [14] explored the use of temporal testers for verification of LTL and PSL. A *temporal tester* for a formula ϕ is a transition system that has a variable x_ϕ such that $\mathbf{G}(x_\phi \leftrightarrow \phi)$ holds; a *positive temporal tester* is similar, except that $\mathbf{G}(x_\phi \rightarrow \phi)$ holds instead. Temporal testers for simple properties can be combined recursively for more complicated properties.

The approach presented in this paper is based on temporal testers. Given a conjunct $\mathbf{G}(z_i \rightarrow \phi)$, ϕ has to be true whenever z_i equals 1. This makes z_i and the monitor state machine for ϕ a positive temporal tester for ϕ.

Similarly, as noted in [13], translation through alternating automata also results in positive temporal testers. The symbolic description of the resulting Büchi automata (depending on the translation method) has a variable for each subformula, with the property that whenever the variable is true, so is the subformula.

C. Finite traces

Some formulas, such as $\mathbf{G}p$ can be shown to hold only by infinite traces, other formulas, like $\mathbf{F}p$, by a finite trace, i.e. the formula will hold on any infinite extension of that finite trace. While, other formulas, such as $\mathbf{G}p \vee \mathbf{F}q$, can sometimes be shown to hold by a finite trace, and in other cases require an infinite trace.

As verification tools are usually much more efficient in detecting finite traces, it is preferable to detect finite traces whenever possible. In *subsection VI-A* it is shown how this can be achieved. The finite traces detected by our method are the same as the *informative prefixes* defined by Kupferman and Vardi in [11]. This is shown in section X.

D. ω-regular specifications

Although this paper shows how to build monitor circuits for LTL and PLTL formulas, Pnueli and Zaks [14] showed how to to extend this method by adding regular events to implement support for ω-regular languages such as PSL or SVA.

III. Notation

By *circuit*, we mean a directed acyclic graph with two edge types, complemented and non-complemented, and the following node/gate types:

AND – A binary AND-gate.
PI – Primary input.
PO – Primary output.
FF – Flip-flop (unit delay).
TRUE – The constant true.

For the main discussion, temporal formulas are expressed in *Linear Temporal Logic* extended with *past operators* (PLTL). The temporal operators of PLTL are reviewed in *Figure 1*. The extension to include past operators is trivial, but it allows us to use a richer set of benchmarks. Detailed formal semantics of PLTL can be found in [3]. A PLTL formula is any expression using logical operators $\wedge$, $\vee$, and $\neg$, and the temporal operators reviewed in *Figure 1*. In our terminology, a *signal* (or *atomic proposition*) is the output of a gate in the design (possibly complemented) referred to by the specification.

ADJACENT STATE

- $\mathbf{X}\,a$ – "next": a holds in the next cycle
- $\mathbf{Y}\,a$ – "yesterday": a held in the previous cycle; FALSE in the first cycle
- $\mathbf{Z}\,a$ – "variant yesterday": same as $\mathbf{Y}$ but TRUE in the first cycle

SIMPLE OPERATORS

- $\mathbf{G}\,a$ – "globally": a holds forever
- $\mathbf{F}\,a$ – "future": a holds at least once in a future (or the current) cycle
- $\mathbf{H}\,a$ – "historically": a a held up to (and including) the current cycle (past dual of $\mathbf{G}$)
- $\mathbf{P}\,a$ – "past": a held at least once in a past (or the current) cycle (past dual of $\mathbf{F}$)

UNTIL OPERATORS

- $[a\ \mathbf{W}\ b]$ – "weak-until": a holds up to the cycle before b holds, or a holds forever
- $[a\ \mathbf{M}\ b]$ – "weak-since": a held since the cycle after b last held, or a held since the first cycle (past dual of $\mathbf{W}$)

$[a\ \mathbf{U}\ b] = [a\ \mathbf{W}\ b] \wedge \mathbf{F}b$ "until"
$[a\ \mathbf{R}\ b] = \neg[\neg a\ \mathbf{U}\ \neg b]$ "release"

$[a\ \mathbf{S}\ b] = [a\ \mathbf{M}\ b] \wedge \mathbf{P}b$ "since" (past dual of $\mathbf{U}$)
$[a\ \mathbf{T}\ b] = \neg[\neg a\ \mathbf{S}\ \neg b]$ "trigger" (past dual of $\mathbf{R}$)

Fig. 1. *Informal overview of the semantics of PLTL operators.*

A PLTL formula is in *negated normal form* (NNF) if negations are present only on the atomic propositions. A formula can be brought into NNF by using the identities $(\neg\mathbf{X}a = \mathbf{X}\neg a)$, $(\neg\mathbf{G}a = \mathbf{F}\neg a)$, $(\neg\mathbf{F}a = \mathbf{G}\neg a)$, $(\neg[a\ \mathbf{U}\ b] = [\neg a\ \mathbf{R}\ \neg b])$, and their past-operator duals. **Example:** $\neg\mathbf{G}(a \vee \mathbf{P}b) = \mathbf{F}(\neg a \wedge \mathbf{H}\neg b)$

IV. On introducing auxiliary variables

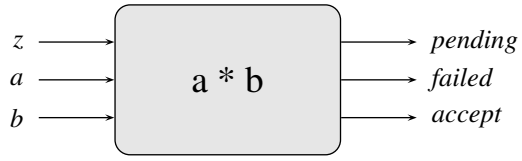
In *Section I*, it was shown how each subformula is given a *name* in the form of an auxiliary variable z_i. The construction is completely analogous to how the Tseitin transformation [15] is used in SAT to convert a propositional formula into an equisatisfiable CNF representation; but because we are dealing with a temporal formula, the $\mathbf{G}$ is needed to ensure that the auxiliary variable maintains its correspondence with the subformula it represents. Because there is an implicit existential quantifier around the LTL formula (*"there-exists-a-path"*), with some abuse of notation we can repeatedly use the identity $\phi(\psi) = \exists x.\mathbf{G}(x \leftrightarrow \psi) \wedge \phi(x)$, but leaving x to be implicitly quantified.

Why is it sound to turn bi-implications into simple implications, as was done from equation (1) to (2)? The operators we

allow in NNF, both logical and temporal, are all monotonically increasing in their inputs, meaning that if $op(x, y)$ is TRUE in a cycle, then so are $op(1, y)$ and $op(x, 1)$. Hence, any trace satisfying equation (2) can be "fixed" by identifying the $z_i \rightarrow$ *RHS* where z_i is 0 and *RHS* is 1 and simply flip the value of z_i. The modified trace will satisfy (1).

V. MONITOR CIRCUITS

Assume the specification has been negated and expanded to an equisatisfiable formula as outlined in *Section I*. Each conjunct is either on the form "$\mathbf{G}\ (z \rightarrow *a)$" (for unary operators "*") or "$\mathbf{G}\ (z \rightarrow a * b)$" (for binary operators). For each operator, we describe a *monitor circuit*. In the next section, it is shown how the monitors are combined to formulate a model checking problem for the entire PLTL specification. Our monitors have the following set of inputs and outputs:



The meanings of these signals are as follows:

z: A fresh PI, also referred to as the *activator*, created to match the auxiliary variable of the expansion. When it non-deterministically goes high, the circuit starts monitoring inputs a and b to see if they adhere to the semantics of the operator.

a: Left input of the operator: either a signal from the design or the activator z_i of the i^{th} monitor, synthesized for the left subformula.

b: Right input of the operator.

pending: TRUE if the monitor has an outstanding requirement on one or both of its input signals to be TRUE either in this or in future cycles.

failed: TRUE if a violation has been detected, preventing any further extension of the current trace from being a valid witness.

accept: Must hold infinitely often for a trace to be a valid witness. Stated negatively: if this signal goes forever FALSE, then the trace is not valid.

The system of monitors can be thought of as follows: The top-monitor is activated by asserting $z_0 = 1$ in the first cycle. This monitor, in order to meet its *accept* condition and avoid its *failed* constraint, will force one or both of its subformulas to be activated, either now or later. The process propagates down through the formula tree. If we can find an infinite run with no monitor outputting *failed*, and with each monitor having an infinite number of *accepts*, then a witness to the temporal formula has been produced. Note that the non-deterministic activator variables are all existentially quantified, which means that we can defer to the underlying model checker to "guess" perfectly when they should be activated.

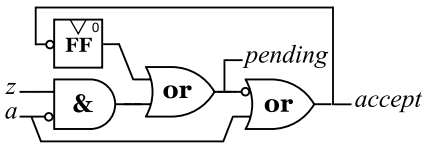


Fig. 2. *Monitor circuit for* $\mathbf{G}(z \rightarrow \mathbf{F}a)$.

Below, we illustrate some LTL operators as monitor circuits. $\mathbf{Y}f$ denotes the previous value of f (which translates directly into a zero-initialized FF whose next-state function is f), and *is_init* denotes a signal which is TRUE only in the first cycle.

If *accept* is left out, it is assumed to be constant TRUE. If *failed* or *pending* are left out, they are assumed to be constant FALSE.

$\mathbf{G}(z \rightarrow \mathbf{X}a)$
$pending = z$
$failed = \mathbf{Y}z \wedge \neg a$

$\mathbf{G}(z \rightarrow \mathbf{G}a)$
$pending = (\mathbf{Y}\ pending) \vee z \qquad [= \mathbf{P}z]$
$failed = pending \wedge \neg a \qquad [= \mathbf{P}z \wedge \neg a]$

$\mathbf{G}(z \rightarrow \mathbf{F}a)$
$pending = (z \vee (\mathbf{Y}\ pending)) \wedge \neg a$
$accept = \neg pending$

$\mathbf{G}(z \rightarrow [a\ \mathbf{W}\ b])$
$pending = (z \vee (\mathbf{Y}\ pending)) \wedge \neg b$
$failed = pending \wedge \neg a$

The 1-to-1 correspondence between this textual representation and a circuit diagram is illustrated for the $\mathbf{F}$ operator in *Figure 2*.

Past operators $\mathbf{P}a$ and $\mathbf{H}a$ are trivially implemented by a single flop remembering if a has held at least once, or always, in the past:

$$once\_a = a \vee (\mathbf{Y}\ once\_a)$$
$$always\_a = a \wedge \neg(\mathbf{Y}\ \neg always\_a)$$

VI. RUNNING THE MODEL CHECKER

Putting together all the steps of our approach:

1) The original specification ϕ is converted to an equivalent NNF formula ψ.
2) ψ is expanded to an equisatisfiable conjunction of "$\mathbf{G}(z_i \rightarrow \langle expr \rangle)$" formulas by introducing a variable z_i for each subformula.
3) For each such conjunct, a monitor circuit is created.
4) The initial activator z_0 is replaced by *is_init*.
5) All *failed* signals are OR-ed together and a flop is introduced to remember if any monitor has ever failed. Formally:

   init(*has_failed*) = 0
   next(*has_failed*) = *FAILED*
   FAILED = $failed_1 \vee \ldots \vee failed_n \vee has\_failed$

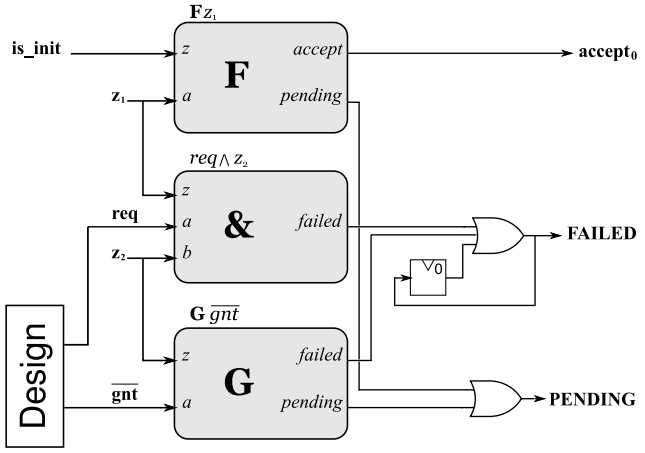


Fig. 3. *Monitor circuit for the LTL formula* $\mathbf{G}(\text{req} \rightarrow \mathbf{F}\,\text{gnt})$. After negation, the formula becomes $\mathbf{F}(req \wedge \mathbf{G}\neg gnt)$, which is translated into: $z_0 \wedge \mathbf{G}(z_0 \rightarrow \mathbf{F}z_1) \wedge \mathbf{G}(z_1 \rightarrow req \wedge z_2) \wedge \mathbf{G}(z_2 \rightarrow \mathbf{G}\neg gnt)$, where activators z_1 and z_2 are two new primary inputs introduced for the subformulas and "is_init", which is true only in the first cycle, replaces z_0 in the circuit.

Which is just another way of stating:

$$FAILED = \mathbf{P}\ (failed_1 \vee \ldots \vee failed_n)$$

6) The liveness checker is called on:

   infinitely_often($accept_1$, …, $accept_n$)

   under the constraint ¬*FAILED*. If the checker does not support constraints, it can be folded into the property:

   infinitely_often($accept_1 \wedge \neg FAILED$, …, $accept_n \wedge \neg FAILED$)

An example of a final monitor circuit is shown in *Figure 3*.

A. Safety fragment

Remember that we are working in the negative, and that disproving a safety property "*for-all-paths:* $\mathbf{G}\ p$" corresponds to finding a witness to "$\mathbf{F}\ \neg p$", i.e. a path to a bad state. Normally witnesses of temporal formulas are infinite traces, but in this case, *any* infinite extension of a finite prefix leading to the bad state is a valid witness. This is a *bounded witness* or *bad prefix* [11], and in our monitor formalism, it corresponds to having *no pending signals*. Therefore, a search for a witness to a temporal formula can be split into two parts: (i) the search for a finite, non-failing trace, where the last state has no pending signal; or (ii) the search for an infinite, non-failing trace where all *accepts* happen infinitely often. The key is that the first type of search can be carried out by a safety-checker, which is more efficient than the more general liveness-checker needed for the second type. The part of the property checkable by (i) is referred to as the *safety fragment*.

This observation can be used to improve our model checking process by:

1) Collecting pending signals:

   $$PENDING = pending_1 \vee \ldots \vee pending_n$$

2) Generating a safety check to be executed before the liveness check:

   reachable($\neg FAILED \wedge \neg PENDING$)

   If this call is UNSAT (no witness found), we run the liveness checker. The liveness property can then be constrained further by adding *PENDING* as a constraint.

B. Assumptions and Assertions

Generally, a specification is composed of two types of formulas, *assumptions*, modeling the behavior of the external environment, and *assertions*, describing the specific behavior of the design under verification. A counterexample for the specification must satisfy all the assumptions and violate at least one of the assertions. Unfortunately, if combined directly into a single LTL formula "*assumptions* → *assertions*", the constraints may force infinite counterexamples where finite ones are expected easier to find. Therefore, most verification tools check safety only under the requirement that assumptions have not yet been violated at the point where the assertion fails. As an example, consider a zero-initialized counter under the assumption "$\mathbf{G}$ (counter < 10)" and the assertion "$\mathbf{G}$ (counter $\neq 5$)". In five cycles, the counter will reach a bad state, but the system has no infinite runs that satisfy the assumption. A safety-checker would produce a counterexample, which is reasonable because the assumption fails after the bad state is reached. In contrast, a liveness tool would consider the property valid because there is no infinite counterexample.

To implement this relaxation in our framework, we ignore *accept* and *pending* for all monitors belonging to assumptions. This clearly changes the semantics of the property, but may be a reasonable compromise (and most probably what the user intended). This can be presented as an option to the user to be accepted or not.

VII. Optimizations

A. Monotonic signals

Suppose the user chose to use past operators to express weak until, as in the right-hand side of the following expression:

$$[a\ \mathbf{W}\ b] \ = \ \mathbf{G}(a \vee \mathbf{P}b)$$

Then, this will lead to the following translation:

$$\begin{array}{rl} & \mathbf{G}(z_0 \ \rightarrow \ \mathbf{G}z_1) \\ \wedge & \mathbf{G}(z_1 \ \rightarrow \ a \vee z_2) \\ \wedge & \mathbf{G}(z_2 \ \rightarrow \ \mathbf{P}b) \end{array}$$

Here we see a problem: as soon as the first monitor is activated ($z_0 = 1$), it will be forever pending. However, the native monitor of weak-until does not share this property. This can be resolved by observing that $\mathbf{P}b$ is a monotonic signal, and that once true, remains true, which motivates introducing a signal *done* for each monitor:

done: This signal should be TRUE only if the monitor has reached a state where *failed* can never happen and *accept*

will hold infinitely often. If this cannot be computed easily, *done* could conservatively be set to FALSE.

The *done* signal can be produced either explicitly by each monitor (extending the contract for what a monitor is), or derived by an analysis of the *failed* and *accept* signals. The pending condition for the $\mathbf{G}$ operator is updated to:

$$\begin{aligned}&\mathbf{G}(z \ \rightarrow \ \mathbf{G}a)\\ &\quad \mathit{pending} = \mathbf{P}z \ \wedge \ \neg a.\mathit{done}\\ &\quad \mathit{failed} = \mathit{pending} \wedge \neg a\end{aligned}$$

The default interpretation of "a.*done*" for a non-activator variable a, is FALSE. But all signals in the specification can be checked for monotonicity in the design by 1-induction, which is typically very fast. If an atomic signal a is monotonically increasing, "a.*done*" can be interpreted as just a. If not, "a.*done*" should still be treated as FALSE.

As an example of how the *done* signal can be explicitly produced, consider the operators:

$$\begin{aligned}&\mathbf{G}(z \ \rightarrow \ \mathbf{P}a)\\ &\quad \mathit{failed} = z \wedge \neg\mathbf{P}a\\ &\quad \mathit{done} = \mathbf{P}a\\ &\mathbf{G}(z \ \rightarrow \ a \vee b)\\ &\quad \mathit{failed} = z \wedge \neg(a \vee b)\\ &\quad \mathit{done} = a.\mathit{done} \ \vee \ b.\mathit{done}\end{aligned}$$

For reasonably sized LTL specifications, we can afford to do the following automated and more precise analysis using symbolic techniques, similar to the constraint analysis of [8]:

Done analysis. For each monitor M_i, let d_i denote "*accept*$\wedge\neg$*failed*" for the signals of that monitor, Let C denote a conjunction of constraints and invariants that known to hold for the system. This will include "$\neg$*FAILED*" as well as "$(\bigwedge_k s_k \ \rightarrow \ s'_k)$" for the monotonic signals $s_1, s_2, \ldots, s_k$ derived from the design. Now, for each monitor, check whether "$d_i \wedge C \rightarrow d'_i$" is true using SAT. If so, let the *done* signal for M_i be defined as d_i and continue the analysis. Optionally, $d_i \rightarrow d'_i$ can be added to C to strengthen future checks.

Note! It should be emphasized at this point that the proposed analysis of *failed* and *accept* signals, as well as the analyses described in the next two subsections, are performed *only* on the combined monitor circuit, which is small, and *not* on the design, which may be large. The only exception is the 1-induction step, which *is* performed on the design and offers a highly selective way of bringing some particularly useful information about the design into the analysis of the monitors. This invariant information (monotonicity of signals) can be used for the *done* analysis above, as well as in the analyses described in the next two subsections.

B. Deadlock states, Acceptable states and Reachable states

Deriving constraints is useful both for strengthening the analyses described in this section, and for proving the property. We make two observations:

- States that for any sequence of inputs will eventually reach *FAILED* cannot be part of a witness.
- States that cannot, for any sequence of inputs, reach a given accept_i signal cannot be part of a witness.

The first type of states corresponds to *deadlock states*, and is characterized by the transitive strong preimage of *FAILED*.[1] This set can be computed symbolically for the combined monitor circuit using e.g. BDDs or SAT based cube-enumeration. The negation is then added as a constraint to the system.

Similarly, the second type of states can be derived by taking the transitive (weak) preimage of each accept_i and intersecting the results. This corresponds precisely to constraint extraction for safety properties as described in [6], interpreting each accept_i signal as a *bad* state.

Finally, the (forward) reachable states of the combined monitors can be computed, and this invariant added to the set of constraints. Although it is redundant in the sense that it will not restrict the search space for finding witnesses, it can make inductive proofs easier and strengthen the analyses presented in this section.

We hypothesize that deriving constraints and invariants will give similar benefits to determinizing the automaton [2] when used with inductive proof-methods, such as k-induction, interpolation and PDR/IC3.

C. Fewer auxiliary variables

Introducing an auxiliary variable for each subformula is not always necessary. It is most obvious for the logical operators, where a subformula with multiple ANDs and ORs can be turned trivially into a single monitor with only one activator z_i, introducing a single new PI.

Also we can save on PIs and get a smaller translation for the $\mathbf{G}$-operator. If we have:

$$\mathbf{G}(z_i \rightarrow \mathbf{G}z_j)$$

then, assuming z_j is a PI introduced for a subformula, we can simply remove it and replace each occurrence of z_j by $\mathbf{P}z_i$. In the same way, we can save up to two PIs for each $\wedge$-operator:

$$\mathbf{G}(z_i \rightarrow z_j \wedge z_k)$$

If z_j and z_k are PIs, they can be replaced by z_i.

These transformations can be understood by looking at the definition of *failed*, which for the $\mathbf{G}$-operator is "$\mathit{failed} = \mathbf{P}z_i \wedge \neg z_j$". Since the left-hand side is constrained to FALSE, we have: $\neg(\mathbf{P}z_i \wedge \neg z_j) \ = \ (\mathbf{P}z_i \rightarrow z_j)$, and since we only need to propagate activation downwards to subformulas, it is safe to substitute z_j for $\mathbf{P}z_i$.

Another way of achieving simplifications of the above sort is by performing *signal correspondence* [16], [4], [12], [9] under constraints. This analysis will detect equivalent nodes in the combined monitor circuit and simplify the netlist by transferring fanouts from all equivalent nodes into one representative node. The fact that these signals are equivalent must

[1] The strong preimage of S is the set of predecessor for which *all* next states are in S.

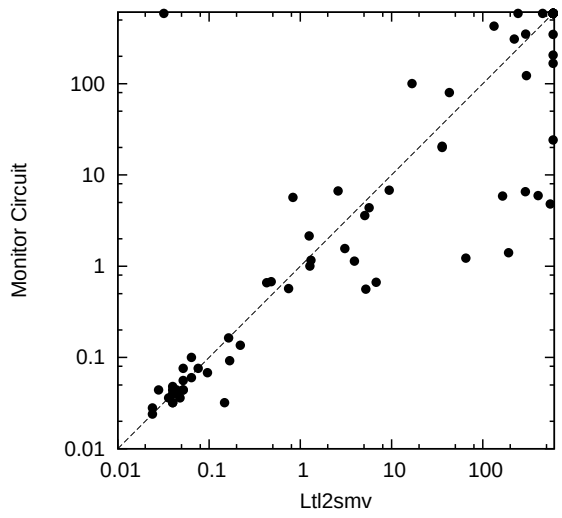


Fig. 4. *Scatter plot of results in log-scale.* This plots include all data points, even the short runs omitted from the table.

be maintained, but if on the left-hand side of the equivalence is a PI (as often happens) with no other fanouts, i.e. "PI $\leftrightarrow$ $\langle$*some node*$\rangle$", then that constraint can always be satisfied and hence dropped.

VIII. Experimental Results

For the experimental evaluation we used the same benchmarks as Biere et. al. in [3], which can be downloaded from [1]. The benchmark suite consists of 14 designs, 12 of which we could use (the *1394* and *csmacd* designs could not be handled correctly by our parser). Each design contains several PLTL properties. For each property, two monitors were generated, one using the method described in this paper, and one using the tool LTL2SMV [7], which builds a monitor from a Büchi automaton produced through the alternating automata approach. Both monitors were then combined with the design and given to a liveness checker [19]. Verification times are reported in *Figure 5* and plotted in *Figure 4*. The benchmarks were carried out on a dual 8-core Intel Xeon E5-2670 with 128 GB of memory, using a timeout of 600 seconds.

Analysis. LTL2SMV provides an alternating automata based approach without much optimizations. This was compared against an unoptimized implementation of the method presented in this paper. Verification runtimes suggests that the two methods are comparable with a small advantage to the new method. Because of its simplicity, this makes it an interesting option for industrial implementation as well as future research.

IX. Acknowledgments

The authors want to thank Shoham Ben-David, Robert Brayton and Alan Mishchenko for their invaluable feedback.

This work is partly supported by SRC contract 2265.001, NSA grant "Enhanced equivalence checking in cryptoanalytic applications", and NSF, grant# 1219154. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Mentor Graphics, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

Design	LTL2SMV (in sec)	CIRCUIT (in sec)
abp4-p2false	6.8	0.7
abp4-p2true	65.3	1.2
abp4-pold	191.6	1.4
abp4-ptimo	5.2	0.6
abp4-ptimoneg	0.8	5.7
bc57-sensors-p0	35.9	20.0
bc57-sensors-p0neg	547.8	4.8
bc57-sensors-p1	0.0	–
bc57-sensors-p1neg	404.6	5.9
bc57-sensors-p2	–	24.2
bc57-sensors-p2neg	164.5	5.9
bc57-sensors-p3	293.1	6.6
brp-p1	43.0	79.9
brp-ptimonegnv	16.8	100.4
dme2-ptimo	5.1	3.6
dme2-ptimonegnv	5.7	4.4
pci-p1	133.0	427.5
pci-pFtimo	9.4	6.8
pci-ptimo	35.8	20.6
prod-cons-p0	1.3	2.1
prod-cons-p0neg	2.6	6.7
prod-cons-p1	0.7	0.6
prod-cons-p1negnv	1.3	1.0
prod-cons-p5	3.1	1.6
prod-cons-p5neg	0.4	0.7
prod-cons-pold1	3.9	1.1
prod-cons-pold3	1.3	1.2
prod-cons-pold4	0.5	0.7
production-cell-p0neg	–	167.2
production-cell-p1	295.1	349.6
production-cell-p1neg	300.6	122.5
production-cell-p2	243.3	–
production-cell-p2neg	–	205.6
production-cell-p3	221.0	308.7
production-cell-p3neg	452.8	–
production-cell-p4	–	347.1
Total solved:	32	33

Fig. 5. *Table of results.* A timeout of 10 minutes was used. Benchmarks that were solved by both approaches in less than 0.5 seconds were removed from the table to conserve space.

References

[1] http://www.tcs.hut.fi/Software/benchmarks/LMCS-2006/.

[2] Roy Armoni, Sergey Egorov, Ranan Fraer, Dmitry Korchemny, and Moshe Y. Vardi. **Efficient LTL compilation for SAT-based model checking**. In *ICCAD*, pages 877–884, 2005.

[3] A. Biere, K. Heljanko, T. Junttila, Latvala T, and V. Schuppan. **Linear Encodings of Bounded LTL Model Checking**. In *Logical Methods in Computer Science, Vol. 2 (5:5)*, pages 1–64, 2006.

[4] P. Bjesse and K. Claessen. **SAT-based Verification without State Space Traversal**. In *Proc. of FMCAD'00. LNCS, Vol. 1954, pp. 372-389.*

[5] Udi Boker, Orna Kupferman, and Adin Rosenberg. **Alternation Removal in Büchi Automata**. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2010.

[6] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. **Speeding up Model Checking by Exploiting Explicit and Hidden Verification Constraints**. In *Proc. of DATE*, 2009.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. **NuSMV Version 2:**

An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[8] Koen Claessen and Niklas Sörensson. **A Liveness Checking Algorithm that Counts**. In *Proc. of FMCAD*, pages 52–59, 2012.

[9] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*, v00127p.

[10] Yonit Kesten, Amir Pnueli, and Li on Raviv. **Algorithmic Verification of Linear Temporal Logic Specifications**. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[11] Orna Kupferman and Moshe Y. Vardi. **Model Checking of Safety Properties**. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1999.

[12] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang. **Scalable and Scalably-verifiable Sequential Synthesis**. In *Proc. of ICCAD'08, pp. 234-241.*

[13] Amir Pnueli and Aleksandr Zaks. **PSL Model Checking and Run-Time Verification Via Testers**. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[14] Amir Pnueli and Aleksandr Zaks. **On the Merits of Temporal Testers**. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer, 2008.

[15] G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.

[16] C. A. J. van Eijk. **Sequential equivalence checking based on structural similarities**. In *IEEE TCAD, 19(7), July 2000, pp. 814-819.*

[17] Moshe Y. Vardi. **Alternating Automata and Program Verification**. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[18] Moshe Y. Vardi and Pierre Wolper. **An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)**. In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[19] Berkeley Verification and Synthesis Research Center (BVSRC). **ABC-ZZ: A C++ framework for verification and synthesis**. *https://bitbucket.org/niklaseen/abc-zz.*

X. Appendix – Informative Prefixes

This section shows that the monitors described in this paper accepts precisely the *informative prefixes*. For brevity, the exposition is limited to the temporal operators $\mathbf{X}$ and $\mathbf{U}$. A *prefix*, or a finite trace, refers to an assignment to the atomic propositions for the first n cycles.

Definition 1 (Kupferman and Vardi [11])**.** *The prefix $s_1, \ldots, s_n$ is **informative** for a formula ϕ if there exists a map L from $\{1, \ldots, n+1\}$ to the set of subformulas of ϕ such that:*

1) $\phi \in L(1)$
2) $L(n+1) = \emptyset$
3) *If an atomic proposition* $p \in L(i)$ *then* $s_i \models p$
4) *If* $a \vee b \in L(i)$ *then* $a \in L(i)$ *or* $b \in L(i)$
5) *If* $a \wedge b \in L(i)$ *then* $a \in L(i)$ *and* $b \in L(i)$
6) *If* $\mathbf{X}a \in L(i)$ *then* $a \in L(i+1)$
7) *if* $[a\ \mathbf{U}\ b] \in L(i)$ *then either* $b \in L(i)$ *or* $a \in L(i)$ *and* $[a\ \mathbf{U}\ b] \in L(i+1)$

To match precisely the formalism of [11], an additional trivial monitor is added for the atomic propositions. For practical purposes it can be optimized away, resulting in the construction described in this paper:

$$\mathbf{G}(z \rightarrow a)$$
$$\mathit{failed} = z \wedge \neg a$$

Proposition 1. *If $s_1, \ldots, s_n$ is an informative prefix for formula ϕ, then the trace $s_1, \ldots, s_n$ is accepted by the monitor circuit of ϕ.*

Proof: The prefix assigns values to the atomic propositions. In our monitor formalism, additional primary inputs are introduced for for the *activator* variables. To complete the trace, the activator for ψ, z_ψ, is set to TRUE in cycle i iff $\psi \in L(i)$. It must now be shown that the augmented trace is accepted by the monitor. More precisely:

(a) The activator of ϕ is TRUE in the initial cycle.
(b) All *failed* signals are FALSE everywhere on the trace.
(c) All *pending* signals are FALSE in cycle n.

We first observe that on this augmented trace, if the *pending* signal holds, the activator must hold as well in the same cycle. For the atomic propositions, boolean connectives and $\mathbf{X}$, the *pending* signal is defined to be the activator signal, so the observation trivially holds. Assume for contradiction that the observation does not hold for $\psi = [a\ \mathbf{U}\ b]$, and let i be the first cycle in which it is false, i.e. the *pending* signal holds but the activator does not. From the monitor constructions, the *pending* signal for $[a\ \mathbf{U}\ b]$ is defined to be:

$$\mathit{pending} = (z_\psi \vee \mathbf{Y}\mathit{pending}) \wedge \neg z_b)$$

which can be simplified by the assumption that $z_\psi = 0$:

$$\mathit{pending} = \mathbf{Y}\mathit{pending} \wedge \neg z_b$$

For *pending* to hold on cycle i, it must hold on cycle $i-1$, but since i is the first cycle in which *pending* can be set without the activator, the activator must be TRUE in cycle $i-1$, and therefore $[a\ \mathbf{U}\ b] \in L(i-1)$. The definition of *pending* also requires that z_b must not hold in cycle $i-1$ and therefore $b \notin L(i-1)$. But the definition of L then forces $[a\ \mathbf{U}\ b]$ to hold in cycle i which contradicts the assumption, proving the observation for $[a\ \mathbf{U}\ b]$. Now:

(a) Follows directly from $\phi \in L(1)$.

(b) We prove for each operator separately:

- For $\psi = p$, an atomic proposition. $\mathit{failed} := z_\psi \wedge \neg p$. However, by definition of L, if $p \in L(i)$ then $s_i \models p$. z_p is true iff $p \in L(i)$, so the combination of $s_i \models \neg p$, and $z_p =$ TRUE can never happen.
- For $\psi = a \wedge b$, $\mathit{failed} := z_\psi \wedge \neg(z_a \wedge z_b)$. However, by definition of L, if $a \wedge b \in L(i)$ then $a \in L(i)$ and $b \in L(i)$. The connectors are TRUE iff the corresponding formulas are in $L(i)$, so the combination of $a \wedge b \in L(i)$, $a \in L(i)$, and $b \in L(i)$, can never happen. The same applies to $\psi = a \vee b$.
- For $\psi = \mathbf{X}a$, $\mathit{failed} := Y z_\psi \wedge \neg z_a$. However, by definition of L, $\mathbf{X}a \in L(i-1)$ and $a \notin L(i)$ can never happen together.
- for $\psi = [a\ \mathbf{U}\ b]$, $\mathit{failed} := \mathit{pending} \wedge \neg z_a$. We proved that if *pending* holds then the activator z_ψ must hold as well, therefore $[a\ \mathbf{U}\ b] \in L(i)$. Substituting the definition of *pending* gives us $\mathit{failed} := (z_\psi \vee \mathbf{Y}\mathit{pending}) \wedge \neg z_b \wedge \neg z_a$. So for failed to be TRUE, we must also have $a, b \notin L(i)$, but cannot happen together with $[a\ \mathbf{U}\ b] \in L(i)$.

(c) For each operator that can possibly set *pending* to TRUE:

- For $\psi = \mathbf{X}a$, $\mathbf{X}a \in L(n)$ requires that $a \in L(n+1)$, and therefore the activator of $\mathbf{X}a$ cannot be set on cycle n. Therefore, neither can *pending*.
- For $\psi = [a\ \mathbf{U}\ b]$, $\mathit{pending} := (z_\psi \vee Y\mathit{pending}) \wedge \neg z_b$. to make the *pending* signal TRUE in cycle n, we must have $[a\ \mathbf{U}\ b] \in L(n)$, which requires that either $a \in L(n) \wedge [a\ \mathbf{U}\ b] \in L(n+1)$ or $b \in L(n)$. The former cannot happen because $L(n+1) = \emptyset$, and therefore z_b must hold, setting the *pending* signal to FALSE.

■

Proposition 2. *If the finite trace $s_1, \ldots, s_n$ is accepted by the monitor circuit of ϕ, then $s_1, \ldots, s_n$ is also an informative prefix for formula ϕ.*

Proof: We need to show the existence of a map L. To facilitate its definition, we change all the activator variables for formulas $[a\ \mathbf{U}\ b]$ in cycle i to TRUE if the *pending* signal holds in cycle $i-1$. This is allowed because activator variables are not atomic proposition, and are not used in the definition of an informative prefix.

If *pending* held at cycle i before the change, then the change does not affect *failed* or *pending*. If *pending* held at cycle $i-1$ and did not hold in cycle i before the change, it means that z_b must already have held in cycle i, so again, *failed* and *pending* are not affected and the trace is still accepted by the monitor.

We define for $i \in \{1, \ldots, n\}$, $L(i) := \{\psi \mid$ activator signal of ψ is set in cycle $i\}$ and $L(n+1) := \emptyset$. It remains to show that this L satisfies the requirements of definition 1. The construction of the monitor guarantees that the activator of ϕ holds in the initial cycle.

The only operators that can prevent $L(n+1)$ from being empty are $\mathbf{X}$ and $\mathbf{U}$. For $\psi = \mathbf{X}a$, the activator cannot hold in cycle n because it is equal to *pending*, therefore $\mathbf{X}a \notin L(n)$. If $[a\ \mathbf{U}\ b] \in L(n)$, and *pending* is FALSE, then z_b must hold on cycle i, and therefore $b \in L(n)$, consistent with $L(n+1) = \emptyset$.

The rest of the conditions also hold. For atomic propositions, the boolean connectives and $\mathbf{X}$, *failed* becomes TRUE when the activator holds and the condition of L is violated. It is only for $\psi = [a\ \mathbf{U}\ b]$ that this is not immediately obvious. If the activator of $[a\ \mathbf{U}\ b]$ holds, then either z_b holds, and *pending* becomes FALSE, or z_b is FALSE and *pending* becomes TRUE, forcing the activator to hold in the next cycle and forcing z_a to hold for *failed* not to become TRUE. This is exactly the condition of definition 1. ■

XI. APPENDIX – LIST OF MONITORS FOR PLTL

Below is a complete list of monitors for PLTL. Variable t is local to each monitor. If *accept* is left out, it is assumed to be constant TRUE. If *failed* or *pending* are left out, they are assumed to be constant FALSE.

$\mathbf{G}(z \rightarrow \mathbf{X}a)$
$\mathit{pending} = z$
$\mathit{failed} = \neg \mathit{is\_init} \wedge (\mathbf{Y}z \wedge \neg a)$

$\mathbf{G}(z \rightarrow \mathbf{Y}a)$
$\mathit{failed} = z \wedge \neg\mathbf{Y}(a)$

$\mathbf{G}(z \rightarrow \mathbf{Z}a)$
$\mathit{failed} = z \wedge \mathbf{Y}(\neg a)$

$\mathbf{G}(z \rightarrow \mathbf{F}a)$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg a$
$\mathit{accept} = \neg \mathit{pending}$

$\mathbf{G}(z \rightarrow \mathbf{G}a)$
$\mathit{pending} = (\mathbf{Y}\ \mathit{pending}) \vee z$
$\mathit{failed} = \mathit{pending} \wedge \neg a$

$\mathbf{G}(z \rightarrow \mathbf{P}a)$
$t = \mathbf{Y}(t) \vee a$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow \mathbf{H}a)$
$t = \neg\mathbf{Y}(\neg t) \wedge a$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow [a\ \mathbf{W}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg b$
$\mathit{failed} = \mathit{pending} \wedge \neg a$

$\mathbf{G}(z \rightarrow [a\ \mathbf{U}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg b$
$\mathit{failed} = \mathit{pending} \wedge \neg a$
$\mathit{accept} = \neg \mathit{pending}$

$\mathbf{G}(z \rightarrow [a\ \mathbf{R}\ b])$
$\mathit{pending} = (z \vee (\mathbf{Y}\ \mathit{pending})) \wedge \neg a$
$\mathit{failed} = (z \wedge \neg b) \vee ((\mathbf{Y}\mathit{pending}) \wedge \neg b)$

$\mathbf{G}(z \rightarrow [a\ \mathbf{S}\ b])$
$t = (\mathbf{Y}t \wedge a) \vee b$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow [a\ \mathbf{T}\ b])$
$t = b \wedge (\neg\mathbf{Y}(\neg t) \vee a)$
$\mathit{failed} = z \wedge \neg t$

$\mathbf{G}(z \rightarrow a \wedge b)$
$\mathit{failed} = z \wedge \neg(a \wedge b)$

$\mathbf{G}(z \rightarrow a \vee b)$
$\mathit{failed} = z \wedge \neg(a \vee b)$

$\mathbf{G}(z \rightarrow$ FALSE$)$
$\mathit{failed} = z$

$\mathbf{G}(z \rightarrow$ TRUE$)$
(nothing)

Invariants for Finite Instances and Beyond

Sylvain Conchon*† Amit Goel‡ Sava Krstić‡ Alain Mebsout*† Fatiha Zaïdi*

*LRI, Université Paris Sud CNRS, Orsay F-91405
†INRIA Saclay – Ile-de-France, Orsay cedex, F-91893
‡Strategic CAD Labs, Intel Corporation

***Abstract*—Verification of safety properties of concurrent programs with an arbitrary numbers of processes is an old challenge. In particular, complex parameterized protocols like FLASH are still out of the scope of state-of-the-art model checkers. In this paper, we describe a new algorithm, called BRAB, that is able to automatically infer invariants strong enough to prove a protocol like FLASH. BRAB computes over-approximations of backward reachable states that are checked to be unreachable in a finite instance of the system. These approximations (candidate invariants) are then model checked together with the original safety properties. Completeness of the approach is ensured by a mechanism for backtracking on spurious traces introduced by too coarse approximations.**

I. Introduction

Nowadays, modern computing systems are often relying on multi-core or distributed architectures. Inherently, the verification of mutual exclusion or cache coherence properties for such systems is very challenging. Consider for instance that, in the Stanford FLASH multiprocessor architecture, the transition system describing the cache coherence protocol has already more than 67 million states when just four processors are in competition [9].

A standard way of verifying a transition system is to enumerate the entire state space [22], [24] (modulo reduction and compaction techniques). However, on large problems, efficient enumerative model checkers reach their limits in both time and memory consumption. For instance, Murφ fails to prove the safety of FLASH for five processes with a timeout limit of 24 hours and 20 GB of memory.

An alternative is to verify a parameterized version of the system. Model checking of such systems is an old and well studied problem [5], [12], [18]. Yet, all automatic techniques that allow properties to be verified for any number of processes do not scale very well. For instance, state-of-the-art parametric model checkers hit their limit on academic problems: most tools need several minutes to prove the safety of the parameterized protocol given by German [36], although this protocol only has 28,000 reachable states for four processes.

Some approaches are known to scale on large problems: compositional and abstraction model checking techniques [10] have been used to prove a parameterized version of FLASH [30], [38]. However, they all require human experts to provide hand-crafted invariants. Designing algorithms to find automatically good quality invariants is still a challenge and an active research area [13], [21], [28], [31], [36].

In this paper, we propose a novel algorithm that infers invariants capable of proving complex protocols. Our contributions are as follows:

- The BRAB algorithm (illustrated in Section II). It first computes a set $\mathcal{M}$ of reachable states using a forward exploration for a finite instance of the system with a *fixed number* of processes. Then, it performs a backward reachability analysis of the *parameterized* system. At each loop iteration, BRAB computes an over-approximation of backward reachable states and checks that it represents states that are not in $\mathcal{M}$. All these approximations, which can be seen as candidate invariants, are model checked together with the original safety properties. To ensure completeness, BRAB backtracks when it encounters a spurious trace introduced by a too coarse approximation.
  The strength of our method resides in two aspects. First, model checking approximations together makes it possible for the proof of an approximation to use part of the proof of another one. A second key insight is that finite instances (even small) are generally good oracles for guiding the choice of approximations as they can be seen as a concentrated knowledge of the system.
- A formalization of BRAB for a generic symbolic framework where sets of states are represented by logical formulas (Section III). We only require pre- and post-images to be computable and the decidability of backward reachability. Under these conditions, we prove soundness, completeness and termination of our algorithm. Such a generic presentation allows BRAB to be implemented in different frameworks.
- An implementation of BRAB in the framework of array-based transition systems (Section IV). This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [19]. Our implementation is available in the Cubicle model checker [14].
- A comparison of our approach with state-of-the-art parameterized and enumerative model checkers on a set of significant problems (Section V). This comparison effort demonstrates that our method is promising.

To our knowledge, Cubicle (with BRAB) is the first tool that proves automatically the safety of FLASH.

II. Invariants for Finite Instances and Beyond

We illustrate our method on a simplified version of the directory based cache coherence protocol proposed by German [36]. The protocol consists of a global directory which maintains the consistency of a shared memory between a

ISBN 978-0-9835678-3-7/13.

parameterized number of cache clients. The status of each cache i is indicated by a variable $\texttt{Cache}[i]$ which can be in one of the three states: (E)xclusive (read and write accesses), (S)hared (read access only) or (I)nvalid (no access to the memory). Clients send requests to the directory when cache misses occur: rs for a shared access (read miss), re for an exclusive access (write miss). The directory has four variables: a boolean flag Exg indicates whether a client has an exclusive access to the main memory, a boolean array Shr, such that $\texttt{Shr}[i]$ is true when a client i is granted (read or write) access to the memory, Cmd stores the current request (ϵ stands for the absence of request), and Ptr contains the emitter of the current request.

The initial states of the protocol are represented by the following logical formula

$$\forall i.\ \texttt{Cache}[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}[i] \ \wedge\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \epsilon$$

stating that the cache of each process is invalid, no access has been given and there is no request to be processed.

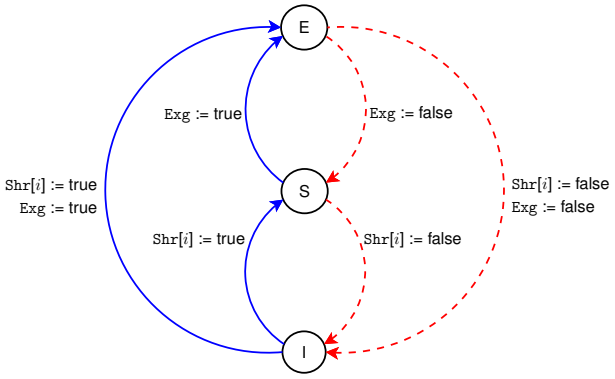


Fig. 1: State diagram of the German-*ish* protocol

We give in Figure 1 a high-level view of the evolution of a single cache. Solid arrows show the evolution of the cache following its own requests whereas, dashed arrows depict transitions resulting from a request of another client. For example, a cache moves from state I to S when a read miss occurs and the directory grants it a shared access, while recording it in the array $\texttt{Shr}[i] := \mathsf{true}$. Similarly, when a write miss occurs in another cache, the directory invalidates all clients recorded in Shr before granting an exclusive access. This has the effect of moving caches from states E or S to state I.

The formal description of the protocol is given by the transition system in Figure 2. Following notations in [36], we describe each transition by a logical formula relating the values of state variables before and after the transition. We denote by $\texttt{X}'$ the value of the variable X after the execution of the transition. For instance, transition t_1 should read as: if there exists a process i whose cache is invalid and there is no command to be processed, then update variable Ptr to i and set variable Cmd to rs.

This protocol ensures that when a cache client is in an exclusive state then no other process has (read or write) access to the memory. Proving this safety property amounts

$$t_1 : \exists i.\ \begin{array}{l}\texttt{Cache}[i] = \mathsf{I} \ \wedge\ \texttt{Cmd} = \epsilon \ \wedge \\ \texttt{Ptr}' = i \ \wedge\ \texttt{Cmd}' = \mathsf{rs}\end{array}$$

$$t_2 : \exists i.\ \begin{array}{l}\texttt{Cache}[i] \neq \mathsf{E} \ \wedge\ \texttt{Cmd} = \epsilon \ \wedge \\ \texttt{Ptr}' = i \ \wedge\ \texttt{Cmd}' = \mathsf{re}\end{array}$$

$$t_3 : \exists i.\ \begin{array}{l}\texttt{Shr}[i] \ \wedge\ \texttt{Cmd} = \mathsf{re} \ \wedge \\ \neg\texttt{Exg}' \ \wedge\ \texttt{Cache}'[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}'[i]\end{array}$$

$$t_4 : \exists i.\ \begin{array}{l}\texttt{Shr}[i] \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \texttt{Exg} \ \wedge \\ \neg\texttt{Exg}' \ \wedge\ \texttt{Cache}'[i] = \mathsf{S}\end{array}$$

$$t_5 : \exists i.\ \begin{array}{l}\texttt{Ptr} = i \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \neg\texttt{Exg} \ \wedge \\ \texttt{Cmd}' = \epsilon \ \wedge\ \texttt{Shr}'[i] \ \wedge\ \texttt{Cache}'[i] = \mathsf{S}\end{array}$$

$$t_6 : \exists i.\ \begin{array}{l}\texttt{Ptr} = i \ \wedge\ \texttt{Cmd} = \mathsf{re} \ \wedge\ \neg\texttt{Exg} \ \wedge\ \forall j.\ \neg\texttt{Shr}[j] \\ \texttt{Cmd}' = \epsilon \ \wedge\ \texttt{Exg}' \ \wedge\ \texttt{Shr}'[i] \ \wedge\ \texttt{Cache}'[i] = \mathsf{E}\end{array}$$

Fig. 2: German-*ish* transition system

to checking that states that satisfy the following formula Θ are not reachable:

$$\Theta :\ \exists i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$$

Finite Instance. We consider a finite instance of the protocol with two caches. We give in Figure 3 (left graph) the beginning of a forward exploration starting from the state (circled node) obtained by instantiating the initial formula with two distinct processes #1 and #2. Each edge label $t(\#_i)$ stands for the instance of a transition t with process $\#_i$.

Backward Reachability. We then run a backward reachability analysis for the parameterized system. Starting from Θ (octagon node), we iteratively compute its pre-images (circle nodes) for all transitions. Pre-images that are subsumed by already visited nodes (dotted edges) are not expanded anymore. This process ends either when a formula in a node intersects the initial formula or when there is no more pre-image to compute.

To improve this standard backward analysis, we try to prune the search space by finding over-approximations of pre-images. Since the set of possibilities is very large, we restrict the choice to formulas that represent unreachable states in the finite instance, and which are syntactic sub-formulas of pre-images.

If it succeeds to extract an appropriate candidate, the newly found approximation (rectangles connected with a bold dashed arrow) replaces the original formula. To illustrate this we show a partial graph of BRAB's execution on the right of Figure 3.

Starting from the unsafe formula $\exists i \neq j.\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$, the pre-image by transition t_5 returns the node $\exists i \neq j.\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \texttt{Ptr} = j \ \wedge\ \texttt{Cache}[i] = \mathsf{E}$. This node could be approximated by $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$. But on closer inspection, $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$ is already reachable on the left graph of Figure 3 as it is satisfied by a concrete state of the finite instance (double-headed arrow with $\models$) so it is undoubtedly not a good approximation. On the contrary

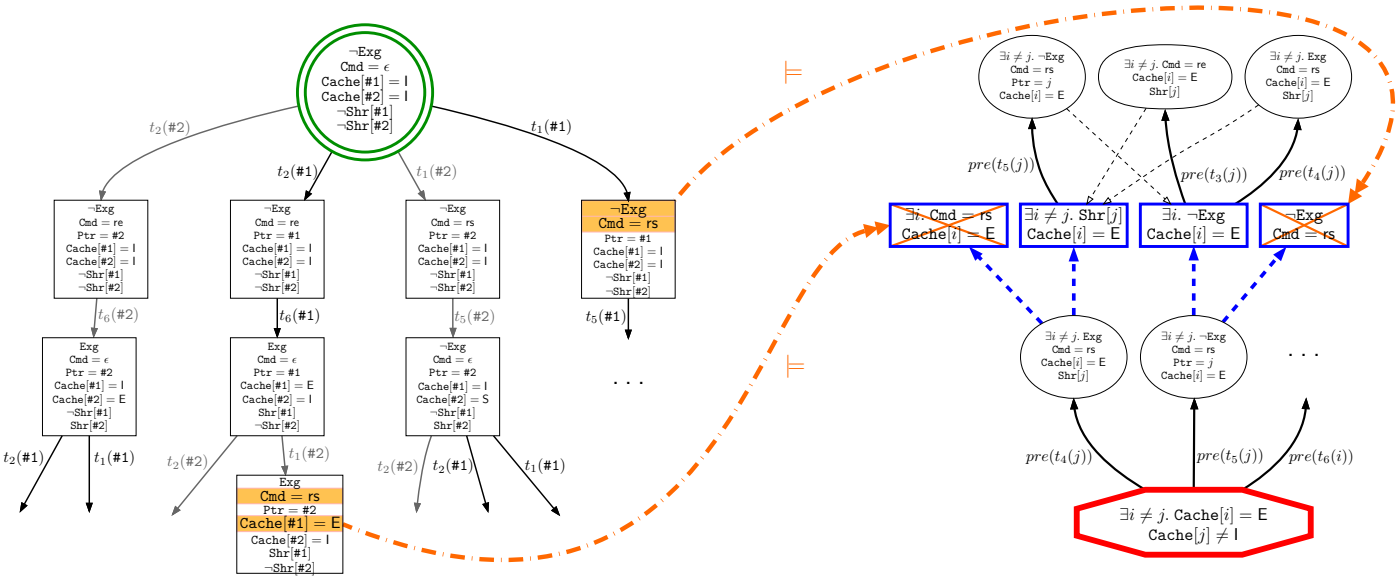


Fig. 3: BRAB on the German-*ish* protocol

no instances of $\exists i.\ \neg\texttt{Exg} \ \wedge\ \texttt{Cache}[i] = \texttt{E}$ is reachable on the finite system of Figure 3. This approximation is inserted in the backward reachability loop which continues as usual.

As we can see on the graph Figure 3, the sub-graphs of some approximations intersect. These sub-graphs depict the proof of unreachability for each approximation, this means that proofs are factorized, hence the benefit of inserting them during the main search. For example, part of the pre-image of the first approximation is subsumed by the second approximation (and *vice-versa*).

Naturally approximations can introduce spurious traces. When one is exposed, BRAB restarts from scratch the construction of the reachability graph, while remembering this approximation to avoid exploring the same spurious behaviours. For example, if we had built the finite model using only one process variable, nothing would prevent the algorithm from considering the bad approximation $\exists i.\texttt{Cmd} = \texttt{rs} \wedge \texttt{Cache}[i] = \texttt{E}$

In our example, with a finite model with two processes, no approximation introduces an error trace and the system is proved safe. As a consequence, each node in the graph is unreachable and its negation is an invariant of the system. In particular, approximations yield the following non-trivial invariants (the last one is not shown on the graph):

$$I_1\colon\quad \forall i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \texttt{E} \implies \neg\texttt{Shr}[j]$$
$$I_2\colon\quad \forall i.\ \texttt{Cache}[i] = \texttt{E} \implies \texttt{Exg}$$
$$I_3\colon\quad \forall i.\ \texttt{Cache}[i] \neq \texttt{I} \implies \texttt{Shr}[i]$$

III. Formalizing the BRAB Algorithm

A. Notations and Preliminaries

We assume the usual syntactic and semantic notions of first-order logic. In particular, we use the symbol $\models$ for the logical entailment relation between sets of formulas. For convenience, disjunctions are represented by sets of formulas.

We adopt a symbolic framework for specification of parameterized systems where states are described by a *fixed set* of state variables $\mathcal{Q}$. Each variable $x \in \mathcal{Q}$ is defined over a finite or infinite domain $\mathcal{D}_x$. This domain may be unspecified, in which case we call it a parameter of the system. We assume that in this framework (sets of) system states can be described by formulas in a decidable fragment of the first-order logic.

A parameterized system S is defined by a pair $(I, \mathcal{T})$ where I is a formula describing the initial states of the system and $\mathcal{T}$ is a set of (possibly quantified) formulas, called *transitions*, relating states of S. For a state formula φ and a transition $\tau \in \mathcal{T}$, let $pre(\tau, \varphi)$ be the formula describing the set of states from which a state satisfying φ can be reached in one τ-step. The pre-image *closure* of φ, denoted by $\textsc{Pre}^*(\varphi)$, is defined as follows

$$\begin{cases} \textsc{Pre}^0(\varphi) & \triangleq \ \varphi \\ \textsc{Pre}^n(\varphi) & \triangleq \ \bigcup\{pre(\tau, \psi) \mid \psi \in \textsc{Pre}^{n-1}(\varphi), \tau \in \mathcal{T}\} \\ \textsc{Pre}^*(\varphi) & \triangleq \ \bigcup_{k\in\mathbb{N}} \textsc{Pre}^k(\varphi) \end{cases}$$

and the pre-image of a set of formulas V is defined by $\textsc{Pre}^*(V) = \bigcup_{\varphi\in V} \textsc{Pre}^*(\varphi)$. We also write $\textsc{Pre}(\varphi)$ for $\textsc{Pre}^1(\varphi)$. Similarly, we define the post-image $post(\tau, \varphi)$ of φ with respect to τ as the set of states that are reachable from φ in one step by taking the transition τ. The definition of $\textsc{Post}^*$ is given by the equations for $\textsc{Pre}^*$, with *post* in place of *pre*. For our purpose, we assume $\textsc{Pre}$ to be effectively computable and $\textsc{Post}$ to be effectively computable on finite instances.

Definition 1. *A set of formulas V is said to be* reachable *iff $\textsc{Post}^*(I) \wedge V$ is satisfiable, or equivalently, $\textsc{Pre}^*(V) \wedge I$ satisfiable. It is* unreachable *otherwise.*

Definition 2. *An* invariant *of a system is any property that holds in all reachable states of the system.*

We give a standard backward reachability algorithm for this framework, as defined by the function `BWD` in Algorithm 1. Starting with an empty set $\mathcal{V}$ of *visited nodes* (state formulas/set really) and a queue $\mathcal{Q}$ of *pending nodes* initialized

with a formula Θ, BWD iteratively computes the backward reachability graph of $\text{PRE}^*(\Theta)$. The algorithm terminates when a node fails the *safety check* (consistency with the initial condition — line 6), or when all nodes in $\mathcal{Q}$ are *subsumed* by the nodes in $\mathcal{V}$ (line 8).

The decidability of BWD is assumed to be guaranteed in the symbolic framework under consideration.

Algorithm 1: Backward Reachability Analysis

Input: a parameterized system $(I, \mathcal{T})$ and a formula Θ
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue

```
1  function BWD() : begin
2      V := ∅;
3      push(Q, Θ);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if (I ∧ φ sat) then
7              return unsafe
8          else if (φ ⊭ V) then
9              V := V ∪ {φ};
10             push(Q, PRE(φ));
11         end
12     end
13     return safe
14 end
```

B. The BRAB Algorithm

BRAB is defined by Algorithms 2 and 3. It implements an extended version of backward reachability that computes over-approximations during the search loop and backtracks when spurious traces are introduced by too coarse formulas.

BRAB takes as input a parameterized system $(I, \mathcal{T})$, a formula Θ, and two integers $d_{\max}$ and k. In addition to the set $\mathcal{V}$ of visited nodes and the work queue $\mathcal{Q}$, our algorithm requires three variables $\mathcal{M}$, $\mathcal{B}$ and $\mathcal{F}$, and a couple of maps Kind and From. These variables are used as follows:

- $\mathcal{M}$ is a set of reachable states for a finite instance of the system with k processes;
- $\mathcal{B}$ is a set of bad (or too coarse) over-approximations;
- $\mathcal{F}$ contains the last visited node that fails the safety check during the backward analysis;
- Kind maps formulas to values in $\{\text{Orig}, \text{Appr}\}$. It is used to differentiate formulas in $\text{PRE}^*(\Theta)$ (Orig formulas) from pre-images of over-approximations (Appr formulas);
- From associates pre-images with their original ancestor formula.

The entry point of the algorithm is the function BRAB. It starts by initializing some variables : $\mathcal{B}$ is the empty set, Θ is recorded as the initial value of $\text{PRE}^*(\Theta)$ in Kind and From, and $\mathcal{M}$ is the set $\text{FWD}(d_{\max}, k)$ of reachable states constructed by a forward exploration of the reachability graph starting in $I(\#1) \wedge \cdots \wedge I(\#k)$ and limited to depth $d_{\max}$. BRAB then enters a verification loop (line 36). It first calls the function BWDA which verifies the safety of Θ in the parameterized case by a backward reachability with approximations. If BWDA returns *safe*, so does BRAB. Otherwise, $\mathcal{F}$ contains the last formula that fails the safety check in BWDA and BRAB returns *unsafe* if $\mathcal{F}$ is a pre-image of Θ. If $\mathcal{F}$ is (a pre-image of) an approximation, then BRAB ignores this spurious result, saves the original ancestor of $\mathcal{F}$ in $\mathcal{B}$ to avoid reproducing the same trace, and continues its verification loop.

BWDA implements a reachability loop similar to Algorithm 1. It only differs at two lines. It saves in $\mathcal{F}$ the formula which fails the safety check (line 23). It also calls function Approx in place of PRE (line 27) to find over-approximations of the current node φ. The function Approx limits potential candidates to subformulas subsuming φ that are not already known to be bad approximations and which represent states that are not in $\mathcal{M}$. If it fails to find such an approximation, Approx returns the pre-image of φ. Regarding the correctness of BRAB, the set candidates(φ) must be finite (implementation details are given in Section IV). If Approx finds a new approximation ψ, it is tagged with Appr in Kind and From(ψ) is set to ψ. Otherwise, formulas in $\text{PRE}(\varphi)$ inherit the information of φ in Kind and From.

C. Correctness

The correctness of BRAB relies on the decidability of BWD (assumed in Section III-A) the following loop invariants:

(Inv1) $\mathcal{V}$ does not contain *immediately* reachable formulas, *i.e.* $\mathcal{V} \models \neg I$

(Inv2) $\text{PRE}^*(\Theta)$ is *incrementally* computed in $\mathcal{V}$ and $\mathcal{Q}$, *i.e.* $\text{PRE}^*(\Theta) \models \mathcal{V} \vee \text{PRE}^*(\mathcal{Q})$

(Inv3) if $\text{Kind}(\varphi) = \text{Orig}$ then $\varphi \in \text{PRE}^*(\Theta)$

Theorem 1. *If* BRAB() *returns* safe *then* Θ *is unreachable.*

Proof: When BRAB returns *safe*, the loop (line 29) terminates with $\mathcal{Q}$ empty. Now, by contradiction, suppose Θ is reachable. By definition $\text{PRE}^*(\mathcal{V}) \wedge I$ is satisfiable. Since $\mathcal{Q}$ is empty, by (Inv2) $\text{PRE}^*(\Theta) \models \mathcal{V}$ and $\mathcal{V} \wedge I$ thus satisfiable too, which contradicts the invariant (Inv1) $\mathcal{V} \models \neg I$. ■

Theorem 2. *If* BRAB() *returns* unsafe *then* Θ *is reachable*

Proof: If BRAB returns *unsafe*, then there exists a formula φ such that $\varphi \wedge I$ is satisfiable and $\text{Kind}(\varphi) = \text{Orig}$. By (Inv3), we conclude that $\text{PRE}^*(\Theta) \wedge I$ is satisfiable. ■

Theorem 3. BRAB() *always returns* safe *or* unsafe

Proof: Suppose the algorithm 2 does not terminate then whether:

1) BWDA does not terminate, or
2) BRAB does not terminate

(1) Since BWDA only differs from BWD by Approx, its termination is assured by the fact that candidates returns a finite set of formuals. (2) The co-domain of From is a subset of

Algorithm 2: Backward Reachability with Approximations and Backtracking (BRAB)

Input: a parameterized system $(I, \mathcal{T})$, a formula Θ to be verified, the maximal depth $d_{\max}$ for the forward exploration and the number k of processes to be considered for the finite instance of the system
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue
$\mathcal{M}$: Finite model obtained by forward exploration
$\mathcal{B}$: bookkeeping of bad approximations
$\mathcal{F}$: last node visited when unsafety discovered
`Kind`: map formulas $\longmapsto$ {Orig, Appr}
`From`: map formulas $\longmapsto$ formula

```
1  function Approx(φ) : begin
2      foreach ψ in candidates(φ) do
3          if ψ ∉ B ∧ M ⊭ ψ then
4              Kind(ψ) := Appr;
5              if Kind(φ) = Orig then From(ψ) := ψ
6              else From(ψ) := From(φ) ;
7              return ψ
8          end
9      end
10     foreach ψ in PRE(φ) do
11         Kind(ψ) := Kind(φ);
12         From(ψ) := From(φ)
13     end
14     return PRE(φ)
15 end
16
17 function BWDA() : begin
18     V := ∅;
19     push(Q, Θ);
20     while not_empty(Q) do
21         φ := pop(Q);
22         if (I ∧ φ sat) then
23             F := φ;
24             return unsafe
25         else if (φ ⊭ V) then
26             V := V ∪ {φ};
27             push(Q, Approx(φ))
28         end
29     end
30     return safe
31 end
32
33 function BRAB() : begin
34     B:= ∅; Kind(Θ) := Orig; From(Θ) := Θ;
35     M := FWD(d_max,k);
36     while BWDA() = unsafe do
37         if Kind(F) = Orig then return unsafe;
38         B := B ∪ {From(F)};
39     end
40     return safe
41 end
```

Algorithm 3: Finite and Depth-Limited Forward Analysis

Input: a parameterized system $(I, \mathcal{T})$
Variables:
$\mathcal{V}$: visited nodes
$\mathcal{Q}$: work queue

```
1  function FWD(d_max,k) : begin
2      V := ∅;
3      push(Q, (0, I(#1) ∧ ··· ∧ I(#k)));
4      while not_empty(Q) do
5          (d, φ) := pop(Q);
6          if (φ ∉ V and d ≤ d_max) then
7              V := V ∪ {φ};
8              N := {(d+1, ψ) | ψ ∈ POST(φ)};
9              push (Q, N)
10         end
11     end
12     return V
13 end
```

$\mathcal{A} = \bigcup_{\varphi \in \mathcal{V}_f}$ `candidates(φ)` (guaranteed by line 5), where $\mathcal{V}_f$ is the final set obtained by `BWD` on Θ. Since $\mathcal{V}_f$ is finite, $\mathcal{A}$ is also finite. The condition $\psi \notin \mathcal{B}$ at line 3 guarantees that approximations added in $\mathcal{B}$ at line 38 are always distinct and in $\mathcal{A}$. In other words $\mathcal{B}$ cannot grow forever, so `BRAB` terminates. ■

Remark Notice that the correctness of BRAB does not depend on the content of $\mathcal{M}$, which thus acts as an oracle.

IV. Implementation

We have implemented BRAB in the logical framework of array-based transition systems that was proposed by Ghilardi and Ranise [19]. In this framework, states are represented by infinite arrays indexed by processes. This class is useful to model several infinite state systems and allows some topology constraints to be specified on indexes (linear ordering, multisets). Although this framework does not have all the desirable properties of Section III-A for completeness, Theorem 1 is still applicable.

In this framework, unsafe properties are cubes (conjunctions of literals existentially quantified by distinct variables). Safety properties are decidable by backward reachability as soon as a well-quasi ordering can be exhibited on models (configurations). The interested reader is referred to [19] for further details. We present here the choices we made for this implementation and its practical aspects.

In `FWD`, the construction of $\mathcal{M}$ only relies on the implementation of POST. Computing POST symbolically could be advantageous for some problems but we found out that a forward *enumerative* exploration worked best (efficiency wise) on our benchmarks. This forces us to abstract away all variables living in unbounded domains and can lead to a model where unreachable states were explored. Since the correctness does not depend on the finite model in any way, its construction can incorporate any state-of-the-art enumerative techniques or methods tailored to bug finding. For instance, its precision could be improved by adding a way of restricting

unbounded types (*e.g.* to handle infinite systems with arithmetic operations). In any case, the only significant quality of the partial model is to be able to disprove the majority of wrong approximations.

In `Approx`, to ensure termination of BRAB, we restrict `candidates(`φ`)` to a finite set of strict syntactic subformulas of φ. In Cubicle, nodes of the proof graph are cubes, seen as sets of literals, so when looking for an approximation, we successively test all its subsets starting from the coarsest ones, *i.e.* the ones that represent the largest sets of states. Choosing first the most general approximations will yield stronger invariants. We keep the first that is not disproved by $\mathcal{M}$ or the set $\mathcal{B}$. Notice this forbids us to directly approximate an approximation. In some cases, no suitable approximation can be found, and we continue the algorithm as usual. Going further than our implementation of `candidates`, for instance to consider all formula ψ such that $\psi \models \varphi$, is possible but is a complicated matter as there exists infinitely many such ψ. In addition, the framework guarantees that pre-images of cubes are computable as disjunctions of cubes.

The efficiency of BRAB relies essentially on two aspects: the choice of approximations and the content of the set $\mathcal{M}$. Indeed, choosing an approximation that is not general enough will delay the convergence of the algorithm and inserting a too general approximation will lead to unnecessary restarts. Similarly, if the exploration of the finite instance is incomplete or too imprecise, some wrong approximation will not be detected before the backward reachability exposes an error trace, resulting in a costly restart. In our implementation, we limit the negative effect of restarts trying to do them as early as possible, by finding wrong approximations sooner. For example, it is a wise choice to give a higher priority in the queue to pre-images of approximations (*i.e.* ψ such that `kind(`ψ`)` $=$ Appr) so as to ensure they will be checked before expanding too much of the original formulas. Instead of restarting the algorithm from scratch, a possible improvement that we did not implement, is to keep as much information as possible from the previous run. It is indeed possible, yet costly, to maintain dependency information at run time to retain the nodes of $\mathcal{V}$ and $\mathcal{Q}$ that are not affected by the wrong approximation. BRAB is distributed in the open source model checker Cubicle.

V. Experimental results

A. Experiments

We have evaluated our implementation of the BRAB algorithm on challenging mutual exclusion algorithms, fault-tolerance and cache coherence protocols. In the table Figure 4, we compare the performance of Cubicle when using a classical backward reachability loop (second column), and when using BRAB (first column). We run BRAB with an unlimited depth forward exploration for two processes in all benchmarks excepted for the different versions of FLASH. We also include the results for an enumerative model checker CMurphi 5.4.9 [35] and different parameterized model checkers (MCMT 2.0 [20], PFS [23], Undip [37]) to show that the examples we chose are far from trivial.

For each tool we report the execution times obtained with the best setting we found. T.O. indicates a timeout if a tool didn't answer within 24 hours and O.M. means that it exceeded a memory limit of 20 GB. For CMurphi, we give the time used to prove the safety of each benchmark for a fixed number of processes between parentheses. The last columns gives the maximum number of processes we were able to reach with 20 GB. We denote by / benchmarks that we were unable to easily translate due to syntactic restrictions. For instance PFS does not allow the update of multiple local variables at the same time, Undip does not allow variables of the type of processes and MCMT doesn't support systems with more than 50 transitions or multi-dimensional arrays.

All benchmarks were executed on a 64 bits machine with an Intel® Xeon® processor @ 3.2 GHz and 24 GB of memory. The source code for Cubicle and its implementation of the BRAB algorithm as well as all the detailed benchmarks are available at `http://cubicle.lri.fr/fmcad2013`.

In this table, Szymanski_at (resp. Szymanski_na) is an atomic (resp. non-atomic) version of Szymanski's mutual exclusion algorithm. German_Baukus is a version of Steven German's protocol extracted from [7]. German.CTC is the version translated from Ching-Tsun Chou's Murφ models, adding data paths to German_Baukus. German_pfs is an encoding of the German that was extracted from the distribution of PFS [23] where invalidation is performed non atomically. Chandra-Toueg is a version of Chandra and Toueg's reliable broadcast protocol [8] with the send omission failures model extracted from [4].

These experiments show that BRAB is very efficient on examples from the literature and is several orders of magnitude faster than backward reachability on almost all benchmarks.

B. The FLASH Cache Coherence Protocol

The Stanford FLASH (FLexible Architecture for SHared memory) multiprocessor [27] is a modular architecture designed to scale to thousands of processing units. Each processor maintains a local cache memory, whose coherence is ensured by a message passing protocol on a point-to-point network with arbitrary latency. Each memory address is *owned* by a processing unit called Home (the physical location of the given memory address).

Related work. The first proof was performed by Park and Dill [34] in 1996 using the PVS proof assistant but it requires to construct inductive invariants by hand and detail the proof steps in the assistant. This protocol was also verified by Das, Dill and Park [15] using a manually guided predicate abstraction. The method of *compositional model checking* and *data type reduction* developed by McMillan [30] which first relied on BDD based model checking in SMV was later elaborated by Chou, Mannava and Park in a framework called CMP [9]. The CMP method, formalized by Krstiċ [26], works by iteratively *abstracting* a protocol and *strengthening* the invariants from the analysis of counterexamples produced by the model checker. As of today, it is the method that scales best but it requires a lot of expert knowledge and manual intervention to devise *non interference lemmas* from counterexamples. In 2008, Talupur and Tuttle came up with the insight of using message flows conceived by protocol designers as a source of potential invariants to help the CMP method converge faster [33], [38]. Although their method is able to automatically

	BRAB	Cubicle	MCMT	PFS	Undip	CMurphi		
Szymanski_at	**0.14s**	0.30s	0.29s	T.O.	32.1s	8.04s (8)	5m12s (10)	2h50m (12)
Szymanski_na	**0.19s**	T.O.	/	/	/	0.88s (4)	8m25s (6)	7h08m (8)
German_Baukus	**0.25s**	7.03s	33m15s	/	9m43s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	**0.29s**	3m23s	T.O.	/	/	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	**0.34s**	3m58s	5m58s	36m05s	T.O.	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	**2m17s**	2h01m	49m25s	/	/	5.68s (4)	2m58s (5)	1h36m (6)
Flash_nodata	**0.36s**	O.M.	/	/	/	4.86s (3)	3m33s (4)	2h46m (5)
Flash	**5m40s**	O.M.	/	/	/	1m27s (3)	2h15m (4)	O.M. (5)

Fig. 4: Benchmarks

generate invariant candidates from message flows, the CMP method still requires adding extra hand crafted non interference lemmas to achieve convergence on the German and FLASH protocols.

We have modeled this protocol in Cubicle's input language from the Murφ models by Ching-Tsun Chou that were used in [9], [38]. These models only account for one memory line but they (and their properties) are straightforwardly generalizable to an arbitrary number of memory addresses. The only difference we introduced is that we abstracted away the `Home` processor and for all arrays indexed by `Home`, each occurrence of `A[Home]` was replaced by a global variable `A_home`. The control property we want to verify for FLASH is

$$\forall x, y.\ x \neq y \Rightarrow \\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{CacheState}[y] \neq \mathsf{Exclusive}$$

and the data properties are

$$\forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{Data}[x] = \texttt{Currdata} \\ \forall x.\ \texttt{CacheState}[x] = \mathsf{Shared} \ \wedge\ \texttt{Collecting} \\ \quad \Rightarrow \texttt{Data}[x] = \texttt{PrevData} \\ \forall x.\ \texttt{CacheState}[x] = \mathsf{Shared} \ \wedge\ \neg\texttt{Collecting} \Rightarrow \\ \quad \texttt{Data}[x] = \texttt{CurrData}$$

where `CurrData`, `PrevData` and `Collecting` are auxiliary variables introduced only to specify these data properties.

We show the results obtained with Cubicle ran with option `-brab` on different versions of the FLASH protocol in Figure 5. The line nodata gives the result when we only asked to verify control properties whereas in line Flash we asked to verify both control and data properties. Finally we give the results on a version of FLASH where we introduced an error in line buggy. On this version, Cubicle exhibits an error trace highlighting a buggy behaviour.

	Forward				Backward			Total
	k	$d_{\max}$	$\lvert\mathcal{M}\rvert$	time	$\lvert\mathcal{V}\rvert$	# inv	$\lvert\mathcal{B}\rvert$	time
nodata	2	6	439	0.02s	37	30	0	**0.36s**
buggy	2	6	445	0.02s	228	/	0	**2.97s**
Flash	3	14	452,523	8.54s	1047	131	0	**5m40s**

Fig. 5: Verification of FLASH with Cubicle

Below are a few of the invariants inferred for the FLASH: they are not trivial although each one of them connects the values of only two or three variables.

$$\begin{array}{ll} (\mathrm{Inv}_1) & \neg\texttt{Invmarked}[\texttt{Home}] \\ (\mathrm{Inv}_2) & \texttt{CacheState}[\texttt{Home}] = \mathsf{Shared} \Rightarrow \\ & \quad (\texttt{Dir\_Local} \vee \neg\texttt{Dir\_Pending}) \\ (\mathrm{Inv}_3) & \forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \texttt{Dir\_Dirty} \\ (\mathrm{Inv}_4) & \forall x.\ \texttt{CacheState}[x] = \mathsf{Exclusive} \Rightarrow \\ & \quad (x = \texttt{Home} \Rightarrow \neg\texttt{Dir\_HeadVld}) \ \wedge \\ & \quad (x \neq \texttt{Home} \Rightarrow \texttt{Dir\_HeadVld}) \end{array}$$

VI. Related Work

Parameterized verification being a largely studied problem, we focus here on the most closely related work.

Invariant generation: Ghilardi and Ranise describe in [19] an invariant synthesis algorithm for array-based systems. Their backward reachability analysis always computes precise formulas and their mechanism of guessing candidate invariants guided by the goal is similar to `BWDA` but the candidates are only filtered by syntactic heuristics. The main difference with our approach is that candidates are model checked one at a time in a completely independent resource limited backward reachability loop. Other approaches for generating inductive invariants include network invariants [21] which uses finite automata learning algorithms and split invariants [32] which connects small-model properties, inductive methods and compositional reasoning.

Cutoffs: The method of invisible invariants [6], [36] aims at discovering inductive invariants for parameterized systems that are checked up to a certain *cutoff* value obtained with a syntactic criterion. Similarly to our approach, it extracts information from a forward exploration of a finite instance of the original system. This information is generalized and, contrary to our technique, must amount to inductive invariants, whereas we only use it as an oracle. Although in Section II $I_1 \wedge I_2 \wedge I_3 \wedge \neg\Theta$ is an inductive invariant, it is generally not the case. In [17], finite instances are also used in conjunction with a template mechanism to obtain formulas that describe interesting system behaviors. Approaches based on *cutoff* and *small model* properties have been most successful when the value is detected dynamically such as in [25] although their method only works for petri-nets, and most recently in [3] which is capable of handling multiple process topologies (arrays, rings, trees, multisets) whereas our implementation of BRAB for array based transition systems only applies for linear topologies (and multisets) but scales for the Flash.

Abstraction: Abdulla *et al.* propose in [1], [2] versions of backward reachability analysis with approximated transitions. Other methods for parameterized verification are based on abstraction: the method of indexed predicates [28] automatically infers quantified predicated from which the technique of

predicate abstraction is able to construct inductive invariants: the tool UCLID which implements this technique is able to verify the German protocol [29] but not the FLASH, counter abstraction [16] whose idea is to keep track of the number of processes that satisfy a given property, and environment abstraction [11] which combines predicate abstraction with counter abstraction. In our case, we do not abstract the original system, abstractions are performed on the fly.

VII. Conclusion and Perspectives

We have presented a novel backward reachability algorithm with approximations and backtracking to check safety properties of parameterized systems. Given a correct backward reachability algorithm, we have proved the correctness of our extension. We believe that small instances of the original problem already exhibit behaviors that constitute a valuable source of knowledge. Our algorithm uses this information to filter approximations which are then model checked altogether, allowing a factoring of the proofs. It can be seen as a technique for automatically inferring invariants. We provide an open source implementation BRAB and have demonstrated the viability of our approach on several examples from the literature and FLASH, a near industrial cache coherence protocol.

An immediate line of future work is to experiment this approach on real industrial protocols such as Intel's LCP or hierarchical cache coherence protocols. While satisfactory, we think that the backtracking mechanism can be improved and that other oracles can be used for the exploration of the finite instance. Finally we would also like to explore the idea of approximations guided by finite instances in other frameworks.

Acknowledgment

This work was partially supported by the French ANR project ANR-12-INSE-0007 Cafein.

References

[1] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*. Springer, 2007.

[2] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*. Springer, 2007.

[3] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI*, pages 476–495, 2013.

[4] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Automated support for the design and validation of fault tolerant parameterized systems: a case study. *ECEASST*, 35, 2010.

[5] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, May 1986.

[6] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234. Springer, 2001.

[7] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI*, pages 317–330. Springer, 2002.

[8] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Distributed algorithms*, pages 289–303. Springer, 1991.

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398. Springer, 2004.

[10] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Press, 1989.

[11] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, pages 126–141. Springer, 2006.

[12] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC'86*. ACM.

[13] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *Form. Methods Syst. Des.*, 34(2):104–125, Apr. 2009.

[14] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV*, pages 718–724. Springer, 2012.

[15] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, pages 160–171. Springer, 1999.

[16] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80. IEEE, 1998.

[17] M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *PLDI*, pages 134–145. ACM, 2010.

[18] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, July 1992.

[19] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[20] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[21] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR*, pages 483–497. Springer, 2006.

[22] O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[23] N. B. Henda and A. Rezine. The PFS prototype model checker. http://www.it.uu.se/research/docs/fm/apv/tools/pfs/.

[24] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[25] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659, 2010.

[26] S. Krstić. Parametrized system verification with guard strengthening and parameter abstraction. In *AVIS*, 2005.

[27] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *ISCA*, pages 302–313. IEEE, 1994.

[28] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281. Springer, 2004.

[29] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[30] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME*, pages 179–195. Springer, 2001.

[31] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427. Springer, 2008.

[32] K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, pages 299–313, 2007.

[33] J. W. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *FMCAD*. IEEE, 2009.

[34] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–310. Springer, 1996.

[35] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *STTT*, 6(4):320–341, 2004.

[36] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97. Springer, 2001.

[37] A. Rezine. UNDIP. http://www.it.uu.se/research/docs/fm/apv/tools/undip.

[38] M. Talupur and M. R. Tuttle. Going with the flow: Parameterized verification using message flows. In *FMCAD*, pages 1–8. IEEE, 2008.

Exploring Interpolants

Philipp Rümmer, Pavle Subotić
Department of Information Technology, Uppsala University, Sweden

***Abstract*—Craig Interpolation is a standard method to construct and refine abstractions in model checking. To obtain abstractions that are suitable for the verification of software programs or hardware designs, model checkers rely on theorem provers to find the right interpolants, or interpolants containing the right predicates, in a generally infinite lattice of interpolants for any given interpolation problem. We present a semantic and solver-independent framework for systematically exploring interpolant lattices, based on the notion of *interpolation abstraction*. We discuss how interpolation abstractions can be constructed for a variety of logics, and how they can be exploited in the context of software model checking.**

I. Introduction

Model checkers use abstractions to reduce the state space of software programs or hardware designs, either to speed up the verification process, or as a way of handling infinite state space. One of the most common methods to construct or refine abstractions is *Craig interpolation* [1], a logical tool to extract concise explanations for the infeasibility (safety) of specific paths in a program. To ensure rapid convergence, model checkers rely on theorem provers to find suitable interpolants, or interpolants containing the right predicates, in a generally infinite lattice of interpolants for any given interpolation problem. In the past, a number of techniques have been proposed to guide theorem provers towards good interpolants (see Sect. II for an overview); however, those techniques either suffer from the fact that they require invasive changes to the theorem prover, or from the fact that they operate on a single proof of path infeasibility, and are therefore limited in the range of interpolants that can be produced.

We present a *semantic* framework for systematically exploring interpolant lattices, based on the notion of *interpolation abstraction*. Our approach is *solver-independent* and works by instrumenting the interpolation query, and therefore does not require any changes to the theorem prover. Despite simple implementation, interpolation abstractions are extremely flexible, and can incorporate domain-specific knowledge about promising interpolants, for instance in the form of *interpolant templates* used by the theorem prover. The framework can be used for a variety of logics, including arithmetic domains or programs operating on arrays or heap, and is also applicable for quantified interpolants.

We have integrated interpolation abstraction into the model checker Eldarica [2], which uses recursion-free Horn clauses (a generalisation of Craig interpolation) to construct abstractions [3], [4]. Our experiments show that interpolation abstraction can prevent divergence of the model checker in cases that are often considered challenging.

A. Introductory Example

We consider an example inspired by the program discussed in the introduction of [5]. The example exhibits a situation that is generally considered challenging for automatic verifiers:

```
i = 0; x = j;
while (i<50) {i++; x++;}
if (j == 0) assert (x >= 50);
```

To show that the assertion holds, a predicate abstraction-based model checker would construct a set of inductive invariants as Boolean combination of given predicates. If needed, Craig interpolation is used to synthesise further predicates.

In the example, we might consider the path to the assertion in which the loop terminates after one iteration. This path could lead to an assertion violation if the conjunction of assignments and guards on the path (in SSA form) is satisfiable:

$$i_0 \doteq 0 \wedge x_0 \doteq j \wedge i_0 < 50 \wedge i_1 \doteq i_0 + 1 \wedge x_1 \doteq x_0 + 1 \quad (1)$$

$$\wedge\, i_1 \geq 50 \wedge j \doteq 0 \wedge x_1 < 50 \quad (2)$$

It is easy to see that the formula is unsatisfiable, and that the path therefore cannot cause errors. To obtain predicates that prevent the path from being considered again in the model checking process, Craig interpolants are computed for different partitionings of the conjuncts; we consider the case $(1) \wedge (2)$, corresponding to the point on the path where the loop condition is checked for the second time. An interpolant is a formula I that satisfies the implications $(1) \rightarrow I$ and $(2) \rightarrow \neg I$, and that only contains variables that occur in both (1) and (2); a model checker will use I as a candidate loop invariant.

The interpolation problem $(1) \wedge (2)$ has several solutions, including $I_1 = (i_1 \leq 1)$ and $I_2 = (x_1 \geq i_1 + j)$. What makes the example challenging is the fact that a theorem prover is likely to compute interpolants like I_1, recognising the fact that the loop cannot terminate after only one iteration as obvious cause of infeasibility. I_1 does not describe a property that holds across loop iterations, however; after adding I_1 as a predicate, a model checker would have to consider the case that the loop terminates after two iterations, leading to a similar formula $i_2 \leq 2$, and so on. Model checking will only terminate after 50 loop unwindings; in similar situations with unbounded loops, picking interpolants like I_1 will lead to divergence (non-termination) of a model checker.

In contrast, the interpolant I_2 encodes a deeper explanation for infeasibility, the dependency between `i` and `x`, and takes the actual assertion to be verified into account. Since I_2 represents an inductive loop invariant, adding it as predicate will lead to significantly faster convergence of a model checker.

This paper presents a methodology to systematically explore solutions of interpolation problems, enabling a model checker

ISBN 978-0-9835678-3-7/13.

to steer the theorem prover towards interpolants like I_2. This is done by modifying the query given to the theorem prover, through the application of *interpolation abstractions* that capture domain knowledge about useful interpolants. To obtain I_2, we over-approximate the interpolation query (1)∧(2) in such a way that I_1 no longer is a valid interpolant:

$$(i_0 \doteq 0 \wedge x_0 \doteq j' \wedge i_0 < 50 \wedge$$
$$i'_1 \doteq i_0{+}1 \wedge x'_1 \doteq x_0{+}1 \wedge \; x'_1 - i'_1 \doteq x_1 - i_1 \wedge j' \doteq j \,)$$
$$\wedge \, (\, x_1 - i_1 \doteq x''_1 - i''_1 \wedge j \doteq j'' \; \wedge i''_1 \geq 50 \wedge j'' \doteq 0 \wedge x''_1 < 50)$$

The rewriting consists of two parts: (i) the variables x_1, i_1, j are renamed to x'_1, i'_1, j' and x''_1, i''_1, j'', respectively; (ii) limited knowledge about the values of x_1, i_1, j is re-introduced, by adding the grey parts of the interpolation query. Note that the formula is still unsatisfiable. Intuitively, the theorem prover "forgets" the precise value of x_1, i_1, ruling out interpolants like I_1; however, the prover retains knowledge about the difference $x_1 - i_1$ (and the value of j), which is sufficient to compute relational interpolants like I_2.

The terms $x_1 - i_1$ and j have the role of *templates,* and encode the domain knowledge that linear relationships between variables and the loop counter are promising building blocks for invariants (the experiments Sect. VII illustrate the generality of this simple kind of template). Template-generated abstractions represent the most important class of interpolation abstractions considered in this paper (but not the only one), and are extremely flexible: it is possible to use both template terms and template formulae, but also templates with quantifiers, parameters, or infinite sets of templates.

Templates are in our approach interpreted *semantically*, not *syntactically*, and it is up to the theorem prover to construct interpolants from templates, Boolean connectives, or other interpreted operations. To illustrate this, observe that the templates $\{x_1 - i_1, i_1\}$ would generate *the same* interpolation abstraction as $\{x_1, i_1\}$; this is because the values of $x_1 - i_1, i_1$ uniquely determine the value of x_1, i_1, and vice versa.

B. Contributions and Organisation of this Paper

- The framework of *interpolant abstractions* (Sect. IV);
- A catalogue of interpolation abstractions, in particular interpolation abstractions generated from *template terms* and *template predicates* (Sect. V);
- Algorithms to explore *lattices of interpolation abstractions,* in order to compute a range of interpolants for a given interpolation problem (Sect. VI);
- An experimental evaluation (Sect. VII).

II. Related Work

Syntactic restrictions of considered interpolants [5], [6], for instance limiting the magnitude of literal constants in interpolants, can be used to enforce convergence and completeness of model checkers. This method is theoretically appealing, and has been the main inspiration for the work presented in this paper. In practice, syntactic restrictions tend to be difficult to implement, since they require deep modifications of an interpolating theorem prover; in addition, completeness does not guarantee convergence within an acceptable amount of time. We present an approach that is semantic and more pragmatic in nature; while not providing any theoretic convergence guarantees, the use of domain-specific knowledge can lead to performance advantages in practice.

It has been proposed to use **term abstraction** to improve the quality of interpolants [7], [8]: intuitively, the occurrence of individual symbols in an interpolant can be prevented through renaming. Our approach is highly related to this technique, but is more general since it enables fine-grained control over symbol occurrences in an interpolant. For instance, in Sect. I-A arbitrary occurrence of the variable i_1 is forbidden, but occurrence in the context $x_1 - i_1$ is allowed.

The **strength of interpolants** can be controlled by choosing different interpolation calculi [9], [10], applied to the same propositional resolution proof. To the best of our knowledge, no conclusive results are available relating interpolant strength with model checking performance. In addition, the extraction of different interpolants *from the same proof* is less flexible than imposing conditions already on the level of proof construction; if a proof does not leverage the right arguments why a program path is infeasible, it is unlikely that good interpolants can be extracted using any method.

In a similar fashion, proofs and interpolants can be **minimised** by means of proof transformations [11], [12]. The same comments as in the previous paragraph apply.

Divergence of model checkers can be prevented by combining interpolation with **acceleration**, which computes precise loop summaries for restricted classes of programs [13], [14], [15]. Again, our approach is more pragmatic, can incorporate domain knowledge, but is not restricted to any particular class of programs. Our experiments show that our method is similarly effective as acceleration for preventing divergence when verifying error-free programs. However, in contrast to acceleration, our method does not support the construction of long counterexamples spanning many loop iterations.

Templates have been used to synthesise program invariants in various contexts, for instance [16], [17], [18], and typically search for invariants within a rigidly defined set of constraints (e.g., with predefined Boolean or quantifier structure). Our approach can be used similarly, with complex building blocks for invariants specified by the user, but leaves the construction of interpolants from templates entirely to the theorem prover.

III. Preliminaries

1) Craig interpolation: We assume familiarity with standard classical logic, including notions like terms, formulae, Boolean connectives, quantifiers, satisfiability, structures, models. For an overview, see, e.g., [19]. The main logics considered in this paper are *classical first-order logic* with equality (FOL) and *Presburger arithmetic* (PA), but our method is not restricted to FOL or PA. In the context of SMT, the quantifier-free fragment of FOL, with equality $\doteq$ as only predicate, is usually denoted by EUF.

Given any logic, we distinguish between *logical symbols,* which include Boolean connectives, equality $\doteq$, interpreted functions, etc., and *non-logical symbols*, among others variables and uninterpreted functions. If $\bar{s} = \langle s_1, \ldots, s_n \rangle$ is a list

of non-logical symbols, we write $\phi[\bar{s}]$ (resp., $t[\bar{s}]$) for a formula (resp., term) containing no non-logical symbols other than $\bar{s}$. We write $\bar{s}' = \langle s'_1, \ldots, s'_n \rangle$ (and similarly $\bar{s}''$, etc.) for a list of primed symbols; $\phi[\bar{s}']$ ($t[\bar{s}']$) is the variant of $\phi[\bar{s}]$ ($t[\bar{s}]$) in which $\bar{s}$ has been replaced with $\bar{s}'$.

An interpolation problem is a conjunction $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ over disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of symbols. An *interpolant* is a formula $I[\bar{s}]$ such that $A[\bar{s}_A, \bar{s}] \Rightarrow I[\bar{s}]$ and $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$; the existence of an interpolant implies that $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable. We say that a logic has the *interpolation property* if also the opposite holds: whenever $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable, there is an interpolant $I[\bar{s}]$. For sake of presentation, we only consider logics with the interpolation property; however, many of the results hold more generally.

We represent *binary relations* as formulae $R[\bar{s}_1, \bar{s}_2]$ over two lists $\bar{s}_1, \bar{s}_2$ of symbols, and relations over a vocabulary $\bar{s}$ as $R[\bar{s}, \bar{s}']$. The identity relation over $\bar{s}$ is denoted by $Id[\bar{s}, \bar{s}']$.

With slight abuse of notation, if $\phi[x_1, \ldots, x_n]$ is a formula containing the free variables $x_1, \ldots, x_n$, and $t_1, \ldots, t_n$ are ground terms, then we write $\phi[t_1, \ldots, t_n]$ for the formula obtained by substituting $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$.

2) Statelessness: Some of the results presented in this paper require an additional assumption about a logic:

Definition 1 A logic is called *stateless* if conjunctions $A[\bar{s}] \wedge B[\bar{t}]$ of satisfiable formulae $A[\bar{s}]$, $B[\bar{t}]$ over disjoint lists $\bar{s}, \bar{t}$ of non-logical symbols are satisfiable.

Intuitively, formulae in a stateless logic interact only through non-logical symbols, not via any notion of global state, structure, etc. Many logics that are relevant in the context of verification are stateless (in particular quantifier-free FOL, PA, logics based on the theory of arrays, etc); other logics, for instance full FOL, modal logics, or separation logic can be made stateless by enriching its vocabulary. Statelessness is important in this paper, since we use the concept of *renaming* of symbols to ensure independence of formulae.

3) Lattices: A *poset* is a set D equipped with a partial ordering $\sqsubseteq$. A poset $\langle D, \sqsubseteq \rangle$ is *bounded* if it has a *least element* $\bot$ and a *greatest element* $\top$. We denote the *least upper bound* and the *greatest lower bound* of a set $X \subseteq D$ by $\bigsqcup X$ and $\bigsqcap X$, respectively, provided that they exist. Given elements $a, b \in D$, we say b is a *successor* of a if $a \sqsubseteq b$ but $a \neq b$, and *immediate successor* if in addition there is no $c \in D \backslash \{a, b\}$ with $a \sqsubseteq c \sqsubseteq b$. Elements $a, b \in D$ with $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$ are *incomparable*. An element $a \in X \subseteq D$ is a *maximal element* (resp., *minimal element*) of X if $a \sqsubseteq b$ (resp., $b \sqsubseteq a$) and $b \in X$ imply $a = b$.

A *lattice* $L = \langle D, \sqsubseteq \rangle$ is a *poset* $\langle D, \sqsubseteq \rangle$ such that $\bigsqcup\{a, b\}$ and $\bigsqcap\{a, b\}$ exist for all $a, b \in D$. L is a *complete lattice* if all non-empty subsets $X \subseteq D$ have a least upper bound and greatest lower bound. A complete lattice is bounded by definition. A non-empty subset $M \subseteq D$ forms a *sub-lattice* if $\bigsqcup\{a, b\} \in M$ and $\bigsqcap\{a, b\} \in M$ for all $a, b \in M$.

A function $f : D_1 \to D_2$, where $\langle D_1, \sqsubseteq_1 \rangle$ and $\langle D_2, \sqsubseteq_2 \rangle$ are posets, is *monotonic* if $x \sqsubseteq_1 y$ implies $f(x) \sqsubseteq_2 f(y)$.

IV. Interpolation Abstractions

This section defines the general concept of interpolation abstractions, and derives basic properties:

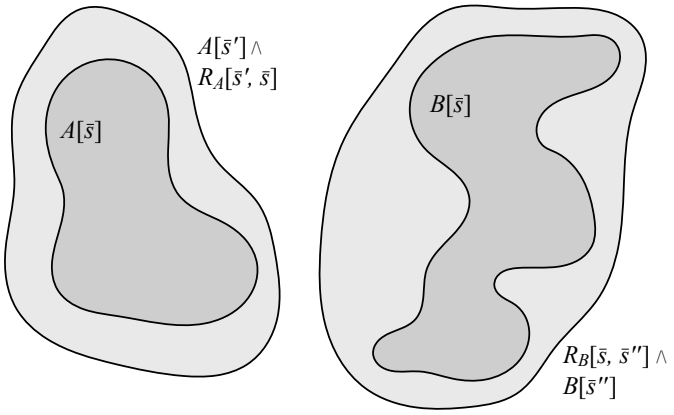


Fig. 1. Illustration of interpolation abstraction, assuming that only common non-logical symbols exist. Both concrete and abstract problem are solvable.

Definition 2 (Interpolation abstraction) Suppose $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are formulae over disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of non-logical symbols, and $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$. An *interpolation abstraction* is a pair $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ of formulae with the property that $R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid (i.e., $Id[\bar{s}', \bar{s}] \Rightarrow R_A[\bar{s}', \bar{s}]$ and $Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']$). We call $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ the *concrete interpolation problem,* and

$$(A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}]) \wedge (R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])$$

the *abstract interpolation problem* for $A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B]$ and $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$.

Assuming that the concrete interpolation problem is solvable, we call an interpolation abstraction *feasible* if also the abstract interpolation problem is solvable, and *infeasible* otherwise.

The common symbols of the interpolation problem in Sect. I-A are $\bar{s} = \langle x_1, i_1, j \rangle$, and the interpolation abstraction is defined by $R_A = (x'_1 - i'_1 \doteq x_1 - i_1 \wedge j' \doteq j)$ and $R_B = (x_1 - i_1 \doteq x''_1 - i''_1 \wedge j \doteq j'')$. A further illustration is given in Fig. 1. The concrete interpolation problem is solvable since the solution sets $A[\bar{s}]$ and $B[\bar{s}]$ are disjoint, i.e., $A[\bar{s}] \wedge B[\bar{s}]$ is unsatisfiable. An interpolant is a formula $I[\bar{s}]$ that represents a superset of $A[\bar{s}]$, but that is disjoint with $B[\bar{s}]$. Since $R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid, the solution set of $A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}]$ represents an *over-approximation* of $A[\bar{s}]$; similarly for $B[\bar{s}]$ and $R_B[\bar{s}, \bar{s}'']$. This ensures the soundness of computed abstract interpolants. In Fig. 1, despite over-approximation, the abstract interpolation problem is solvable, which means that the interpolation abstraction is feasible.

Lemma 3 (Soundness) Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).

Interpolation abstractions can be used to guide interpolation engines, by restricting the space $Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])$ of interpolants satisfying an interpolation problem. For this, recall that the set $Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv$ of interpolant classes (modulo logical equivalence) is closed under conjunctions (meet) and disjunctions (join), so that $(Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv, \Rightarrow)$ is a lattice. Fig. 2 shows the

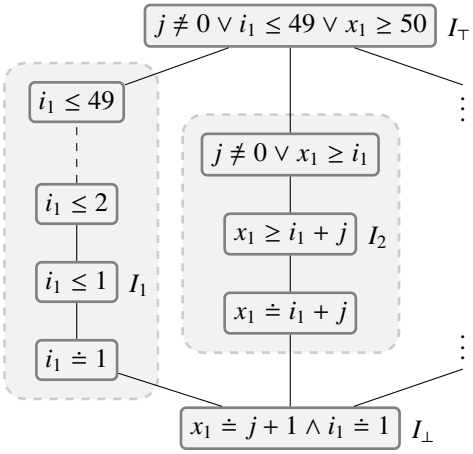


Fig. 2. Parts of the interpolant lattice for the example in Sect. I-A (up to equivalence). The dashed boxes represent the sub-lattices for the abstraction induced by the template terms $\{i_1\}$ (left) and $\{x_1 - i_1, j\}$ (right).

interpolant lattice for the example in Sect. I-A; this lattice has a strongest concrete interpolant $I_\bot$ and a weakest concrete interpolant $I_\top$. In general, the interpolant lattice might be incomplete and not contain such elements.

For a feasible abstraction, the lattice of abstract interpolants

$$(\mathit{Inter}(A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}],\ R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])/\equiv,\ \Rightarrow)$$

is a sub-lattice of the concrete interpolant lattice. The sub-lattice is *convex,* because if I_1 and I_3 are abstract interpolants and I_2 is a concrete interpolant with $I_1 \Rightarrow I_2 \Rightarrow I_3$, then also I_2 is an abstract interpolant. The choice of the relation R_A in an interpolation abstraction constrains the lattice of abstract interpolants from below, the relation R_B from above.

We illustrate two disjoint sub-lattices in Fig. 2: the left box is the sub-lattice for the abstraction $(i_1' \doteq i_1, i_1 \doteq i_1'')$, while the right box represents the interpolation abstraction

$$(x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j,\ x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$$

used in Sect. I-A to derive interpolant I_2.

As the following lemma shows, there are no principal restrictions how fine-grained the guidance enforced by an interpolation abstraction can be; however, since abstraction is a semantic notion, we can only impose constraints *up to equivalence of interpolants:*

Lemma 4 (Completeness) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem with interpolant $I[\bar{s}]$ in a stateless logic, such that both $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are satisfiable (the problem is not degenerate). Then there is a feasible interpolation abstraction (definable in the same logic) such that every abstract interpolant is equivalent to $I[\bar{s}]$.

V. A Catalogue of Interpolation Abstractions

This section introduces a range of practically relevant interpolation abstractions, mainly defined in terms of *templates* as illustrated in Sect. I-A. For any interpolation abstraction, it is interesting to consider the following questions:

(i) provided the concrete interpolation problem is solvable, characterise the cases in which also the abstract problem can be solved (how *coarse* the abstraction is);
(ii) provided the abstract interpolation problem is solvable, characterise the space of abstract interpolants.

The first point touches the question to which degree an interpolation abstraction limits the set of proofs that a theorem prover can find. We hypothesise (and explain in Sect. I-A) that it is less important to generate interpolants with a specific syntactic shape, than to force a theorem prover to use the *right argument* for showing that a path in a program is safe.

We remark that interpolation abstractions can also be combined, for instance to create abstractions that include both template terms and template predicates. In general, the component-wise conjunction of two interpolation abstractions is again a well-formed abstraction, as is the disjunction.

A. Finite Term Interpolation Abstractions

The first family of interpolation abstractions is defined with the help of finite sets T of *template terms,* and formalises the abstraction used in Sect. I-A. Intuitively, abstract interpolants for a term abstraction induced by T are formulae that only use elements of T, in combination with logical symbols, as building blocks (a precise characterisation is given in Lem. 7 below). For the case of interpolation in EUF (quantifier-free FOL without uninterpreted predicates), this means that abstract interpolants are Boolean combinations of equations between T terms. In linear arithmetic, abstract interpolants may contain equations and inequalities over linear combinations of T terms.

The relations defining a term interpolation abstraction follow the example given in Sect. I-A, and assert that primed and unprimed versions of T terms have the same value. As a consequence, nothing is known about the value of unprimed terms that are *not* mentioned in T.

Definition 5 (Term interpolation abstraction) Suppose that $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The interpolation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

$$R_A^T[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}], \quad R_B^T[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} t_i[\bar{s}] \doteq t_i[\bar{s}'']$$

is called *term interpolation abstraction* over T.

Term abstractions are feasible if and only if a concrete interpolant exists that can be expressed purely using T terms:

Lemma 6 (Solvability) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The abstract interpolation problem for $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ is solvable if and only if there is a formula $I[x_1, \ldots, x_n]$ over n variables $x_1, \ldots, x_n$ (and no further non-logical symbols) such that $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant of $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.

Example 1 Consider the interpolation abstraction used in Sect. I-A, which is created by the set $T = \{x_1 - i_1, j\}$ of terms. The abstract interpolation problem is solvable with interpolant

$x_1 \geq i_1 + j$, which can be represented as $(x_1 - i_1) \geq (j)$ as a combination of the template terms in T.

It would be tempting to assume that *all* interpolants generated by term interpolation abstractions are as specified in Lem. 6, i.e., constructed only from T terms and logical symbols. In fact, since our framework restricts the space of interpolants in a semantic way, only weaker guarantees can be provided about the range of possible interpolants; this is related to the earlier observation (Sect. IV) that interpolation can only be restricted *up to logical equivalence:*

Lemma 7 (Interpolant space) Suppose the abstract interpolation problem for $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ is solvable, and the underlying logic is EUF or PA. Then there is a strongest abstract interpolant $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$, and a weakest abstract interpolant $I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$, each constructed only from T terms and logical symbols. A formula $J[\bar{s}]$ is an abstract interpolant iff the implications $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]] \Rightarrow J[\bar{s}] \Rightarrow I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ hold.

Example 2 Again, consider Sect. I-A, and the interpolant lattice as shown in Fig. 2. The strongest abstract interpolant for the interpolation abstraction induced by $T = \{x_1 - i_1, j\}$ is $x_1 \doteq i_1 + j$, the weakest one $j \not\doteq 0 \vee x_1 \geq i_1$.

B. Finite Predicate Interpolation Abstractions

In a similar way as sets of terms, also finite sets of *formulae* induce interpolation abstractions. Template formulae can be relevant to steer an interpolating theorem prover towards (possibly user-specified or quantified) interpolants that might be hard to find for the prover alone. The approach bears some similarities to the concept of predicate abstraction in model checking [20], [21], but still leaves the use of templates entirely to the theorem prover.

Definition 8 (Predicate interpolation abstraction) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $Pred = \{\phi_1[\bar{s}], \ldots, \phi_n[\bar{s}]\}$ is a finite set of formulae. $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ defined by

$$R_A^{Pred}[\bar{s}', \bar{s}] \;=\; \bigwedge_{i=1}^{n} (\phi_i[\bar{s}'] \rightarrow \phi_i[\bar{s}])$$

$$R_B^{Pred}[\bar{s}, \bar{s}''] \;=\; \bigwedge_{i=1}^{n} (\phi_i[\bar{s}] \rightarrow \phi_i[\bar{s}''])$$

is called *predicate interpolation abstraction* over *Pred*.

Intuitively, predicate interpolation abstractions restrict the solutions of an interpolation problem to those interpolants that can be represented as a positive Boolean combination of the predicates in *Pred*. Note that it is possible to include the negation of a predicate $\phi[\bar{s}]$ in *Pred* if *negative* occurrences of $\phi[\bar{s}]$ are supposed to be allowed in an interpolant (or both $\phi[\bar{s}]$ and $\neg\phi[\bar{s}]$ for both positive and negative occurrences).

Lemma 9 (Solvability) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and *Pred* a finite set of predicates. If the underlying logic is stateless, then the abstract interpolation problem for $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ is solvable if and only if $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ has an interpolant $I[\bar{s}]$ that is a positive Boolean combination of predicates in *Pred*.

We remark that the implication $\Leftarrow$ holds in all cases, whereas $\Rightarrow$ needs the assumption that the logic is stateless. As a counterexample for the stateful case, consider the interpolation problem $(\forall x, y.\ x \doteq y) \wedge (\exists x, y.\ x \not\doteq y)$ in full FOL. The abstract interpolation problem is solvable even for $Pred = \emptyset$ (with interpolant $\forall x, y.\ x \doteq y$), but no positive Boolean combination of *Pred* formulae is an interpolant.

The interpolant space can be characterised as for term interpolation abstractions (Lem. 7):

Lemma 10 (Interpolant space) Suppose the abstract interpolation problem for $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ is solvable, and the underlying logic is stateless. Then there is a strongest abstract interpolant $I_\bot[\bar{s}]$, and a weakest abstract interpolant $I_\top[\bar{s}]$, each being a positive Boolean combination of predicates in *Pred*. A formula $J[\bar{s}]$ is an abstract interpolant iff the implications $I_\bot[\bar{s}] \Rightarrow J[\bar{s}] \Rightarrow I_\top[\bar{s}]$ hold.

C. Quantified Interpolation Abstractions

The previous sections showed how interpolation abstractions are generated by finite sets of templates. A similar construction can be performed for *infinite* sets of templates, expressed schematically with the help of variables; in the verification context, this is particularly relevant if arrays or heap are encoded with the help of uninterpreted functions.

Example 3 Suppose the binary function H represents heap contents, with heap accesses *obj.field* translated to $H(obj, field)$, and is used to state an interpolation problem:

$$(H(a, f) \doteq c \wedge H(b, g) \not\doteq null) \wedge$$
$$(b \doteq c \wedge H(b, g) \doteq null \wedge H(H(a, f), g) \doteq null)$$

An obvious interpolant is the formula $I_1 = (H(b, g) \not\doteq null)$. Based on domain-specific knowledge, we might want to avoid interpolants with direct heap accesses $H(\cdot, g)$, and instead prefer the pattern $H(H(\cdot, f), g)$. To find alternative interpolants, we can use the templates $\{H(H(x, f), g), a, b, c\}$, the first of which contains a schematic variable x. The resulting abstraction excludes I_1, but yields the interpolant
$I_2 \;=\; (b \doteq c \rightarrow H(H(a, f), g) \not\doteq null)$.

Definition 11 (Schematic term abstraction) Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, and a finite set $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_1]\}$ of terms with free variables $\bar{x}_1, \ldots, \bar{x}_n$. The interpolation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

$$R_A^T[\bar{s}', \bar{s}] \;=\; \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}', \bar{x}_i] \doteq t_i[\bar{s}, \bar{x}_i],$$

$$R_B^T[\bar{s}, \bar{s}''] \;=\; \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}, \bar{x}_i] \doteq t_i[\bar{s}'', \bar{x}_i]$$

is called *schematic term interpolation abstraction* over T.

Note that schematic term interpolation abstractions reduce to ordinary term interpolation abstractions (as in Def. 5) if none of the template terms contains free variables.

Quantified abstractions are clearly less interesting for logics that admit quantifier elimination, such as PA, but they are relevant whenever uninterpreted functions (EUF) are involved.

Lemma 12 (Solvability in EUF) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem in EUF, $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_1]\}$ a finite set of schematic terms, and $f = \langle f_1, \ldots, f_n \rangle$ a vector of fresh functions with arities $|\bar{x}_1|, \ldots, |\bar{x}_n|$, respectively. The abstract interpolation problem for $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ is solvable if and only if there is a formula $I[f_1, \ldots, f_n]$ (without non-logical symbols other than $\bar{f}$) such that $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ is an interpolant of $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.

The expression $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ denotes the formula obtained by replacing each occurrence of a function f_i in $I[f_1, \ldots, f_n]$ with the template $t_i[\bar{s}, \bar{x}_i]$, substituting the arguments of f_i for the schematic variables $\bar{x}_i$.

VI. Exploration of Interpolants

In practice, given an interpolation problem, we want to compute a whole *range* of interpolants, or alternatively find an interpolant that is optimal with respect to some objective. For instance, in the example in Sect. I-A, we consider interpolant I_2 constructed using templates $\{x_1 - i_1, j\}$ as "better" than interpolant I_1 for the template i_1. To formalise this concept of *interpolant exploration* we arrange families of interpolation abstractions as *abstraction lattices,* and present search algorithms on such lattices. Abstraction lattices are equipped with a monotonic mapping μ to abstractions (R_A, R_B), ordered by component-wise implication. The following paragraphs focus on the case of *finite* abstraction lattices; the handling of infinite (parametric) abstraction lattices is planned as future work.

Definition 13 (Abstraction lattice) Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$. An *abstraction lattice* is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a monotonic mapping μ from elements of $\langle L, \sqsubseteq_L \rangle$ to interpolation abstractions $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ with the property that $\mu(\bot) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$.

The elements of an abstraction lattice that map to *feasible* interpolation abstractions form a downward closed set; an illustration is given in Fig. 3, where feasible elements are shaded in gray. Provided that the concrete interpolation problem is solvable, the set of feasible elements in the lattice is non-empty, due to the requirement that $\mu(\bot) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$.

Particularly interesting are *maximal feasible* interpolation abstractions, i.e., the maximal elements within the set of feasible interpolation abstractions. Maximal feasible abstractions restrict interpolants in the strongest possible way, and are therefore most suitable for exploring interpolants; we refer to the set of maximal feasible elements as *abstraction frontier*.

A. Construction of Abstraction Lattices

When working with interpolation abstractions generated by templates, abstraction lattices can naturally by constructed as the *powerset lattice* of some template base set (ordered by the superset relation); this construction applies both to term and predicate templates. Another useful construction is to form the *product* of two lattices, defining the mapping μ as the conjunction (or alternatively disjunction) of the individual mappings μ_1, μ_2.

Example 4 An abstraction lattice for the example in Sect. I-A is $(\langle \wp(T), \supseteq \rangle, \mu)$, with base templates $T = \{x_1 - i_1, i_1, j\}$ and μ mapping each element to the abstraction in Def. 5. Note that the bottom element of the lattice represents the full set T of templates (the weakest abstraction), and the top element the empty set $\emptyset$ (the strongest abstraction). Also, note that $\mu(T)$ is the identity abstraction $(Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$, since T is a basis of the vector space of linear functions in x_1, i_1, j.

The lattice is presented in Fig. 3, with feasible elements in light gray. The maximal feasible elements $\{i_1\}$ and $\{x_1 - i_1, j\}$ map to interpolation abstractions with the abstract interpolants I_1 and I_2, respectively, as illustrated in Fig 2. Smaller feasible elements (closer to $\bot$) correspond to larger sub-lattices of abstract interpolants, and therefore provide weaker guidance for a theorem prover; for instance, element $\{j, i_1\}$ can produce all abstract interpolants that $\{i_1\}$ generates, but can in addition lead to interpolants like $I_3 = (j \neq 0 \vee i_1 \leq 49)$.

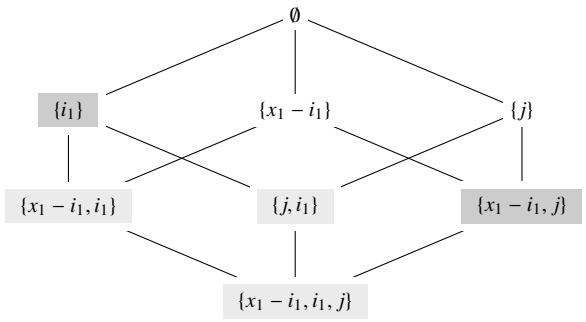


Fig. 3. The abstraction lattice for the running example. The light gray shaded elements are feasible, the dark gray ones maximal feasible.

B. Computation of Abstraction Frontiers

We present an algorithm to compute abstraction frontiers of finite abstraction lattices. The search is described in Algorithms 1 and 2. Algorithm 1 describes the top-level procedure for finding minimal elements in an abstraction lattice. Initially we check if the $\bot$ element is infeasible (line 1). If this is the case, then the concrete interpolation problem is not solvable and we return an empty abstraction frontier. Otherwise, we initialise the frontier with a maximal feasible element (line 4), which is found by the *maximise* function (described in Algorithm 2). Next, in line 5 we check whether a feasible element can be found that is incomparable to all frontier elements found so far; efficient methods for computing such incomparable elements can be defined based on the shape of the chosen abstraction lattice, and are not shown here. As long as incomparable elements can be found, we compute further maximal feasible elements and add them to the frontier.

In Algorithm 2 we describe the procedure for finding a maximal feasible element *mfe* with the property that $elem \sqsubseteq mfe$. In each iteration of the maximisation loop, it is checked whether

Algorithm 1: Exploration algorithm

Input: Interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of maximal feasible interpolation abstractions

```
1 if ⊥ is infeasible then
2 |   return ∅;
3 end
4 Frontier ← {maximise(⊥)};
5 while ∃ feasible elem ∈ L, incomparable with Frontier do
6 |   Frontier ← Frontier ∪ {maximise(elem)};
7 end
8 return Frontier;
```

Algorithm 2: Maximisation algorithm

Input: Feasible element: *elem*
Result: Maximal feasible element

```
1 while ∃ feasible successor fs of elem do
2 |   pick element middle such that fs ⊑_L middle ⊑_L ⊤;
3 |   if middle is feasible then
4 |   |   elem ← middle;
5 |   else
6 |   |   elem ← fs;
7 |   end
8 end
9 return elem;
```

elem has any feasible parents (line 1); if this is not the case, *elem* has to be maximal feasible and is returned. Otherwise, in the loop body the algorithm executes a binary search on the set of elements in between *elem* and $\top$. The algorithm depends on the ability to efficiently compute (random) middle elements between two elements $a \sqsubset b$ of the lattice (line 2); again, this functionality can best be implemented specifically for an individual lattice, and is not shown here.

Lemma 14 (Correctness of exploration algorithm) When applied to a finite abstraction lattice, Algorithm 1 terminates and returns the set of maximal feasible elements.

A useful refinement of the exploration algorithm is to *canonise* lattice elements during search. Elements $a, b \in L$ are considered equivalent if they are mapped to (logically) equivalent abstraction relations by μ. Canonisation can select a representative for every equivalence class of lattice elements, and search be carried out only on such canonical elements.

C. Selection of Maximal Feasible Elements

Given the abstraction frontier, it is possible to compute a range of interpolants solving the original interpolation problem. However, for large abstraction frontiers this may be neither feasible nor necessary. It is more useful to define a measure for the quality of interpolation abstractions, again exploiting domain-specific knowledge, and only use the best abstractions for interpolation.

To select good maximal feasible interpolation abstractions, we define a function $cost : L \to \mathbb{N}$ that maps elements of an abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ to a natural number, with lower values indicating that an interpolation abstraction is considered better. In the case of abstractions constructed using a powerset lattice over templates ($L = \wp(T)$), it is natural to assign a cost to every element in T ($cost : T \to \mathbb{N}$), and to define the cost of a lattice element $A \in L$ as $cost(A) = \sum_{t \in A} cost(t)$.

Our abstraction lattice in Fig. 3 has two maximal feasible elements, $\{i_1\}$ and $\{x_1 - i_1, j\}$, that result in computing the interpolants I_1 and I_2, respectively. We can define a cost function that assigns a high cost to $\{i_1\}$ and a low cost to $\{x_1 - i_1, j\}$, expressing the fact that we prefer to not talk about the loop counter i_1 in absolute terms. More generally, assigning a high cost to variables representing loop counters is a reasonable strategy for obtaining general interpolants (a similar observation is made in [7], and implemented with the help of "term abstraction").

VII. Integration into a Software Model Checker

A. General Integration

Interpolation abstraction can be applied whenever interpolation is used by a model checker to eliminate spurious counterexamples. To this end, it is necessary to select one or multiple *abstraction points* in the constructed interpolation problem (which might concern an inductive sequence of interpolants, tree interpolants, etc.), and then to define an abstraction lattice for each abstraction point. For instance, when computing an inductive sequence $I_0, I_1, \ldots, I_{10}$ for the conjunction $P_1 \wedge \cdots \wedge P_{10}$, we might select interpolants I_3 and I_5 as abstraction points, choose a pair of abstraction lattices, and add abstraction relations to the conjuncts P_3, P_4, P_5, P_6.

We then use Algorithm 1 to search for maximal feasible interpolation abstractions in the Cartesian product of the chosen abstraction lattices. With the help of cost functions, the best maximal feasible abstractions can be determined, and subsequently be used to compute abstract interpolants.

B. Abstraction in Eldarica

We have integrated our technique into the predicate abstraction-based model checker Eldarica [2], which uses Horn clauses to represent different kinds of verification problems [3], and solves recursion-free Horn constraints to synthesise new predicates for abstraction [4]. As abstraction points, recurrent control locations in counterexamples are chosen (corresponding to recurrent relation symbols of Horn clauses), which represent loops in a program. Abstraction lattices are powerset lattices over the template terms

$$\{z \mid z \text{ a variable in the program}\}$$
$$\cup \{x + y,\ x - y \mid x, y \text{ variables assigned in the loop body}\}$$

In Table I we evaluate the performance of our approach compared to Eldarica without interpolation abstraction, the acceleration-based tool Flata [2], and the Horn engine of Z3 [22] (v4.3.2). Benchmarks are taken from [15], and from a recent collection of Horn problems in SMT-LIB format.[1] They tend to be small ($10-750$ Horn clauses each), but challenging

[1] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/

Benchmark	Eldarica		Eldarica-ABS		Flata	Z3
	N	sec	N	sec	sec	sec
C programs from [15]						
boustrophedon (C)	*	*	10	10.7	*	0.1
boustrophedon_expansed (C)	*	*	11	7.7	*	0.1
halbwachs (C)	*	*	53	2.4	*	0.1
gopan (C)	17	22.2	62	57.0	0.4	349.5
rate_limiter (C)	11	2.7	11	19.1	1.0	0.1
anubhav (C)	1	1.7	1	1.6	0.9	*
cousot (C)	*	*	3	7.7	0.7	*
bubblesort (E)	1	2.8	1	2.3	83.2	0.3
insdel (C)	1	0.9	1	0.9	0.7	0.0
insertsort (E)	1	1.8	1	1.7	1.3	0.1
listcounter (C)	*	*	8	2.0	0.2	*
listcounter (E)	1	0.9	1	0.9	0.2	0.0
listreversal (C)	1	1.9	1	1.9	4.9	*
mergesort (E)	1	2.9	1	2.6	1.1	0.2
selectionsort (E)	1	2.4	1	2.4	1.2	0.2
rotation_vc.1 (C)	7	2.0	7	0.3	1.9	0.2
rotation_vc.2 (C)	8	2.7	8	0.2	2.2	0.3
rotation_vc.3 (C)	0	2.3	0	0.2	2.3	0.0
rotation.1 (E)	3	1.8	3	1.8	0.5	0.1
split_vc.1 (C)	18	3.9	17	3.2	*	1.1
split_vc.2 (C)	*	*	18	1.1	*	0.2
split_vc.3 (C)	0	2.8	0	1.5	*	0.0
Recursive Horn SMT-LIB Benchmarks						
addition (C)	1	0.7	1	0.8	0.4	0.0
bfprt (C)	*	*	5	8.3	-	0.0
binarysearch (C)	1	0.9	1	0.9	-	0.0
buildheap (C)	*	*	*	*	-	*
countZero (C)	2	2.0	2	2.0	-	0.0
disjunctive (C)	10	2.4	5	5.0	0.2	0.3
floodfill (C)	*	*	*	*	41.2	0.1
gcd (C)	4	1.2	4	2.0	-	*
identity (C)	2	1.1	2	2.1	-	0.1
mccarthy91 (C)	4	1.4	3	2.4	0.2	0.0
mccarthy92 (C)	38	5.6	7	8.7	0.1	0.1
merge-leq (C)	3	1.1	7	7.0	15.7	0.1
merge (C)	3	1.1	4	4.5	14.7	0.1
mult (C)	*	*	15	52.8	-	*
palidrome (C)	4	1.4	2	2.1	-	0.1
parity (C)	4	1.6	4	2.9	0.8	*
remainder (C)	2	1.1	3	1.6	-	*
running (C)	2	0.9	2	1.7	0.2	0.1
triple (C)	4	2.0	4	5.1	-	0.1

TABLE I

COMPARISON OF ELDARICA WITHOUT INTERPOLATION ABSTRACTION, ELDARICA WITH **ABS**TRACTION, FLATA, AND Z3. THE LETTER AFTER THE MODEL NAME DISTINGUISHES **C**ORRECT BENCHMARKS FROM BENCHMARKS WITH A REACHABLE **E**RROR STATE. FOR ELDARICA, WE GIVE THE NUMBER OF REQUIRED CEGAR ITERATIONS (N), AND THE RUNTIME IN SECONDS; FOR FLATA AND Z3, THE RUNTIME IS GIVEN. ITEMS WITH "*" INDICATE A TIMEOUT (SET TO 10 MINUTES), WHILE - INDICATES INABILITY TO RUN THE BENCHMARK DUE TO LACK OF SUPPORT FOR SOME OPERATORS IN THE PROBLEMS. EXPERIMENTS WERE DONE ON AN INTEL CORE I7 DUO 2.9 GHZ WITH 8GB OF RAM.

for model checkers. We focused on benchmarks on which Eldarica without interpolation abstraction diverges; since interpolation abstraction gives no advantages when constructing long counterexamples, we mainly used correct benchmarks (programs not containing errors). Lattice sizes in interpolation abstraction are typically $2^{15} - 2^{300}$; we used a timeout of 1s for exploring abstraction lattices.

The results demonstrate the feasibility of our technique and its ability to avoid divergence, in particular on problems from [15]. Overall, interpolation abstraction only incurs a reasonable runtime overhead. The biggest (relative) overhead could be observed for the rate_limiter example, where some of the feasibility checks for abstraction take long time. Flata is able to handle a number of the benchmarks on which Eldarica times out, but can overall solve fewer problems than Eldarica. Z3 is able to solve many of the benchmarks very quickly, but overall times out on a larger number of benchmarks than Eldarica with interpolation abstraction.

VIII. CONCLUSION

We have presented a semantic and solver-independent framework for guiding theorem provers towards high-quality interpolants. Our method is simple to implement, but can improve the performance of model checkers significantly. Various directions of future work are planned: (i) develop further forms of interpolation abstraction, in particular quantified and parametric ones; (ii) application of the framework to programs with arrays and heap; (iii) clearly, our approach is related to the theory of abstract interpretation; we plan whether methods from abstract interpretation can be carried over to our method.

Acknowledgements: We thank Hossein Hojjat and Viktor Kuncak for discussions, and for assistance with the implementation in Eldarica. We are also grateful for helpful comments from the referees. This work was supported by the EU FP7 STREP CERTAINTY and by the Swedish Research Council.

REFERENCES

[1] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, 1957.
[2] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A verification toolkit for numerical transition systems - tool paper," in *FM*, 2012, pp. 247–251.
[3] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.
[4] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for Horn-clause verification," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 347–363.
[5] R. Jhala and K. L. McMillan, "A practical and complete approach to predicate refinement," in *TACAS*, 2006, pp. 459–473.
[6] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *TACAS*, 2008, pp. 413–427.
[7] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, "Lazy abstraction with interpolants for arrays," in *LPAR*, 2012.
[8] N. Totla and T. Wies, "Complete instantiation-based interpolation," in *POPL*, R. Giacobazzi and R. Cousot, Eds. ACM, 2013, pp. 537–548.
[9] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *VMCAI*, 2010, pp. 129–145.
[10] S. F. Rollini, O. Sery, and N. Sharygina, "Leveraging interpolant strength in model checking," in *CAV*, 2012, pp. 193–209.
[11] S. Rollini, R. Bruttomesso, and N. Sharygina, "An efficient and flexible approach to resolution proof reduction," in *HVC*, 2010, pp. 182–196.
[12] K. Hoder, L. Kovács, and A. Voronkov, "Playing in the grey area of proofs," in *POPL*, 2012, pp. 259–272.
[13] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun, "Accelerating interpolation-based model-checking," in *TACAS*, 2008, pp. 428–442.
[14] M. N. Seghir, "A lightweight approach for loop summarization," in *ATVA*, 2011, pp. 351–365.
[15] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer, "Accelerating interpolants," in *ATVA*, 2012, pp. 187–202.
[16] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME*, 2001, pp. 500–517.
[17] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Invariant synthesis for combined theories," in *VMCAI*. Springer, 2007.
[18] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *PLDI*, 2009, pp. 223–234.
[19] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
[20] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *CAV*, 1997, pp. 72–83.
[21] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *31st POPL*, 2004.
[22] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2012, pp. 157–171.
[23] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "Beyond quantifier-free interpolation in extensions of Presburger arithmetic," in *VMCAI*, ser. LNCS. Springer, 2011.

Synthesizing Multiple Boolean Functions using Interpolation on a Single Proof

Georg Hofferek[1] Ashutosh Gupta[2] Bettina Könighofer[1] Jie-Hong Roland Jiang[3] Roderick Bloem[1]
[1]Graz University of Technology, Austria [2]IST Austria [3]National Taiwan University

***Abstract*—It is often difficult to correctly implement a Boolean controller for a complex system, especially when concurrency is involved. Yet, it may be easy to formally specify a controller. For instance, for a pipelined processor it suffices to state that the visible behavior of the pipelined system should be identical to a non-pipelined reference system (Burch-Dill paradigm). We present a novel procedure to efficiently synthesize multiple Boolean control signals from a specification given as a quantified first-order formula (with a specific quantifier structure). Our approach uses uninterpreted functions to abstract details of the design. We construct an unsatisfiable SMT formula from the given specification. Then, from just one proof of unsatisfiability, we use a variant of Craig interpolation to compute multiple coordinated interpolants that implement the Boolean control signals. Our method avoids iterative learning and back-substitution of the control functions. We applied our approach to synthesize a controller for a simple two-stage pipelined processor, and present first experimental results.**

I. Introduction

Some program parts are easier to write than others. Freedom of deadlocks, for instance, is trivial to specify but not to implement. These parts lend themselves to synthesis, in which a difficult part of the program is written automatically. This approach has been followed in program sketching [20], [22], [21], in lock synthesis [25], and in synthesis using templates [9], [23], [24].

In this paper, we consider systems that have multiple unimplemented Boolean control signals. The systems that we will consider may not be entirely Boolean. We will consider systems with uninterpreted functions, but our method extends to systems with linear arithmetic. For example, consider a microprocessor. Following Burch and Dill [5], we assume that a reference implementation of the datapath is available. Constructing a pipelined processor is not trivial, as it involves implementing control logic signals that control the hazards arising from concurrency in the pipeline. Correctness of the pipelined processor is stated as equivalence with the reference implementation. In this setting, we can avoid the complexity of the datapath (which is the same in the two implementations) by abstracting it away using uninterpreted functions. Where Burch and Dill verify that the implementation of the control signals is correct, we construct a correct implementation automatically. This problem was previously addressed in [12]. We improve over that paper by directly encoding the problem into SMT, thus avoiding BDDs, and by avoiding backsubstitution in case multiple functions are synthesized.

Our approach is also applicable to synthesis of conditions in (loop-free) programs. As noted in [9], synthesizing loop-free programs can be a building block of full program synthesis. Prior work [20] presented various techniques to deal with finite loops. Those techniques are also applicable in our framework.

To synthesize a single missing signal, we can introduce a fresh uninitialized Boolean variable c. We can express the specification as a logical formula $\forall I \exists c \forall O.\Phi(I, c, O)$, which states that, for each input I, there exists a value of c such that each output O of the function is correct. Here, I and O can come from non-Boolean domains. If an implementation is possible, the formula is valid and a witness function for c is an implementation of the missing signal.

Following [14], we can generate a witness using interpolation. In this paper, we generalize this approach by allowing $n \geq 1$ missing components to be synthesized simultaneously. This leads us to a formula of the form $\forall I \exists c_1 \dots c_n \forall O.\Phi(I, c_1 \dots c_n, O)$. We use an SMT solver to prove a related formula unsatisfiable and use interpolation [18] to obtain the desired witness functions. The first contribution of this paper is to extend prior work [14] beyond the propositional level, and consider formulas expressed in the theory of uninterpreted functions and equality. As a second contribution, we propose a new technique, called *n-interpolation*, which corresponds to simultaneously computing n *coordinated* interpolants from just one proof of unsatisfiability. Like the interpolation procedures of [11], [15], we need a "colorable" proof, which we produce by transforming a standard proof from an SMT solver.

Our algorithm avoids the iterative interpolant computation described in [14], where interpolants are iteratively substituted into the formula. As the iterative approach needs one SMT solver call per witness function, and interpolants may grow dramatically over the iterations, this computation may be costly and may yield large interpolants. A similar back-substitution method is also used in [2] for GR(1) synthesis and in [16] for functional synthesis. Our new method requires the expansion of the the (Boolean) existential quantifier, increasing the size of the formula exponentially (w.r.t. the number of control signals). Note, however, that previous approaches [14] have the same limitation.

II. Illustration

In this Section we illustrate our approach using a simple controller synthesis problem. Figure 1 shows an incomplete

This research was supported by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), the Austrian Science Fund (FWF) through projects RiSE (S11406-N23) and QUAINT (I774-N23), and ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

ISBN 978-0-9835678-3-7/13.

hardware design. There are two input bit-vectors i_1 and i_2, carrying non-zero signed integers, and also two output bit-vectors o_1 and o_2 carrying signed integers. The block neg flips the sign of its input. The outputs are controlled by two bits, c_1 and c_2. The controller of c_1 and c_2 is not implemented. Suppose the specification of the incomplete design states that the signs of the two outputs must be different. Formally, the specification is

$$\forall i_1, i_2.\exists c_1, c_2.\forall o_1, o_2.((c_1 \wedge o_1 = i_1 \vee \neg c_1 \wedge o_1 = neg(i_1)) \wedge (c_2 \wedge o_2 = i_2 \vee \neg c_2 \wedge o_2 = neg(i_2))) \rightarrow (pos(o_1) \leftrightarrow \neg pos(o_2)),$$

where the predicate pos returns $\mathtt{T}$ iff its parameter is positive. We can compute witness functions for c_1 and c_2 using n-interpolation.[1] Our method returns witness functions $c_1 = pos(i_1)$ and $c_2 = \neg pos(i_2)$. (Other functions are also possible.)

Note that computing two interpolants independently may not work. For instance, we may choose $c_1 = \mathtt{T}$ or we can take $c_2 = \mathtt{T}$, but we cannot choose $c_1 = c_2 = \mathtt{T}$. This problem is normally solved by substituting one solution before the next is computed, but our method computes both interpolants simultaneously and in a coordinated way.

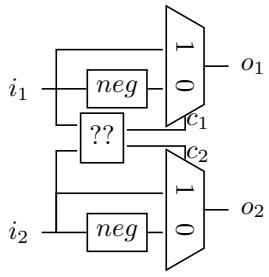


Fig. 1. Example of controller synthesis.

III. Preliminaries

A. Uninterpreted Functions and Arrays

We consider the *Theory of Uninterpreted Functions and Equality* $\mathcal{T}_U$. We have variables $x \in \mathcal{X}$ from an uninterpreted domain, Boolean variables $b \in \mathcal{B}$, uninterpreted function symbols $f \in \mathcal{F}$, and uninterpreted predicate symbols $P \in \mathcal{P}$. The following grammar defines the syntax of the formulas in $\mathcal{T}_u$.

$$\begin{aligned} \text{terms} \ni t &::= x \mid f(t, \ldots, t), \\ \text{atoms} \ni a &::= b \mid P(t, \ldots, t) \mid t = t, \\ \text{formulas} \ni \phi &::= a \mid \neg\phi \mid \phi \vee \phi. \end{aligned}$$

Let $\phi_1 \wedge \phi_2$ be short for $\neg(\neg\phi_1 \vee \neg\phi_2)$. Let $a \neq b$ be short for $\neg(a = b)$. Let $\mathtt{T} = \phi \vee \neg\phi$, let $\mathtt{F} = \neg\mathtt{T}$, and let $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$.

A *literal* is an atom or its negation. Let l be a literal. If $l = \neg a$ then let $\neg l = a$. A *clause* is a set of literals, interpreted as the disjunction. The empty clause $\emptyset$ denotes $\mathtt{F}$. A *conjunctive formula* is the negation of a clause. A *CNF formula* is a set of clauses. A CNF formula is interpreted as the conjunction of its clauses. Since any formula can be converted into a CNF formula, we will assume that all the formulas in this paper are CNF formulas. Let ϕ and ψ be CNF formulas/clauses/literals. Let $Symb(\phi)$ be the set of variables, functions, and predicates occurring in ϕ. Let $\phi \preceq \psi$ iff $Symb(\phi) \subseteq Symb(\psi)$. Let $Lits(\phi) = \{a, \neg a \mid a \text{ is an atom in } \phi\}$. For a clause C, let $C|_\phi = \{s \in C \mid s \preceq \phi\}$.

[1] We must add the axiom $(pos(i_1) \oplus pos(neg(i_1))) \wedge (pos(i_2) \oplus pos(neg(i_2)))$.

$$\textsc{Hyp}\ \frac{}{C}\ C \in \phi,\ \phi \in \text{CNF} \qquad \textsc{Axi}\ \frac{}{C}\ \vdash_{\mathcal{T}_U} C$$

$$\textsc{Res}\ \frac{a \vee C \quad \neg a \vee D}{C \vee D}$$

Fig. 2. Sound and complete proof rules for the theory $\mathcal{T}_U$.

Arrays are useful for modeling memory whose size is not known a priori. We will use a decidable fragment, known as the *Array Property Fragment* with *uninterpreted indices* to create specifications from which we synthesize controllers. Bradley et al. [4] present an algorithm to reduce formulas with array properties to equisatisfiable formulas over the theory of uninterpreted functions. Hofferek and Bloem [12] show that this algorithm generalizes to the quantified formulas that occur in controller synthesis problems. For the rest of this paper, we assume that specifications and formulas containing array properties have been reduced to formulas over the theory of uninterpreted functions.

B. Proofs of Unsatisfiability

We consider the usual semantics of formulas in $\mathcal{T}_u$. The problem of proving unsatisfiability of formulas is decidable. Many *Satisfiability Modulo Theories (SMT) Solvers* exist that can decide the satisfiability of CNF-$\mathcal{T}_U$ formulas, and, in case the formula is not satisfiable, produce a *proof of unsatisfiability*.

A (named) *proof rule* is a template for a logic entailment between a (possibly empty) list of *premises* and a *conclusion*. Templates for premises are written above a horizontal line, templates for conclusions below. Possible conditions for the application of the proof rule are written on the right-hand side of the line.

The proofs we consider will be based on the rules given in Fig. 2. They form a sound and complete proof system for proving unsatisfiability of a CNF-$\mathcal{T}_U$ formula ϕ. The Hyp rule is used to introduce clauses from ϕ into the proof. The Axi rule is used to introduce theory-tautology clauses. In their simplest form, these clauses represent concrete instances of theory axioms (reflexivity, symmetry, transitivity and congruence). However, as our proof transformation algorithms will produce theory tautologies that are based on several axioms, we use the following, less restrictive, definition.

Definition 1 (Theory-Tautology Clause)**.** *A* theory-tautology clause *is a clause of the form* $(\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_k \vee b)$ *that is tautologically true within the theory* $\mathcal{T}_U$*. The literals* $\neg a_i$*, for* $0 < i \leq k$*, are called the* implying literals *and the (positive) literal* b *is called the* implied literal*.*

The Res rule is the standard resolution rule to combine clauses that contain one literal in opposite polarity respectively. We will call this literal the *resolving literal* or the *pivot*.

Definition 2 (Unsatisfiability Proof)**.** *An* unsatisfiability proof *for a CNF-*$\mathcal{T}_U$ *formula* ϕ *is a directed, acyclic graph (DAG)* (N, E)*, where* $N = \{r\} \cup N_I \cup N_L$ *is the set of nodes (partitioned into the root node* r*, the set of internal nodes* N_I*, and the set of leaf nodes* N_L*), and* $E \subseteq N \times N$ *is the set of (directed) edges. Every* $n \in N$ *is labeled with the name of a proof rule* $rule(n)$ *and a clause* $clause(n)$*. The graph has to fulfill the following properties:*

(1) $clause(r) = \emptyset$.
(2) *For all* $n \in N_L$, $clause(n)$ *is either a clause from* ϕ *(if* $rule(n) =$ HYP*) or a theory-tautology clause (if* $rule(n) =$ AXI*).*
(3) *The nodes in* N_L *whose clauses are theory-tautology clauses can be ordered in such a way that for each such node each implying literal either occurs in* ϕ*, or is an implied literals of the tautology clause of a a preceding node (according to the order).*[2]
(4) *The root has no incoming edges, the leaves have no outgoing edges, and all nodes in* $n \in N \setminus N_L$ *have exactly 2 outgoing edges, pointing to nodes* n_1, n_2*, with* $n_1 \neq n_2$*. Using* $clause(n_1)$ *and* $clause(n_2)$ *as premises and* $clause(n)$ *as conclusion must yield a valid instance of proof rule* $rule(n)$.

We used the VERIT SMT solver [3], which provides proofs that conform to these requirements.

C. Transitivity-Congruence Chains

Given a set A of atoms, we can use the well-known congruence-closure algorithm to construct a *congruence graph* [8] according to the following definition.

Definition 3 (Congruence Graph)**.** *A* congruence graph *over a set* A *of atoms is a graph which has terms as its nodes. Each edge is labeled either with an* equality justification*, which is an equality atom from* A *that equates the terms connected by the edge, or with a* congruence justification*. A congruence justification can only be used when the terms connected by the edge are both instances* $f(a_1, \ldots, a_k)$ *and* $f(b_1, \ldots, b_k)$ *of the same uninterpreted function* f*. In this case, the congruence justification is a set of* k *paths in the graph connecting the* a_i *and* b_i *respectively, not using the edge which they label.*

Definition 4 (Transitivity-Congruence Chain)**.** *A* transitivity-congruence chain $\pi = (a \rightsquigarrow b)$ *is a path in a congruence graph that connects terms* a *and* b*. Let* $Lits(\pi)$ *be the set of literals of the path, which is defined as the union of the literals of all edges on the path. The literal of an edge labeled with an equality justification* p *is the set* $\{p\}$*. The set of literals of an edge labeled with a congruence justification with paths* π_i *is recursively defined as* $\bigcup_i Lits(\pi_i)$.

Theorem 1. *The conjunction of the literals in a transitivity-congruence chain* $(a \rightsquigarrow b)$ *implies* $a = b$ *within* $\mathcal{T}_U$*. I.e.,* $(\bigvee_{l \in Lits(a \rightsquigarrow b)} \neg l) \vee (a = b)$ *is a theory-tautology clause.*

D. Craig Interpolation

Let ϕ and ψ be CNF formulas such that $\phi \wedge \psi$ is unsatisfiable. The algorithm presented in [18] for computing an interpolant between ϕ and ψ needs a proof of unsatisfiability of $\phi \wedge \psi$. By annotating this proof with the partial interpolants, the algorithm computes the interpolant. In this paper, we present slightly different annotation rules to compute interpolants, which are results of mixing ideas from [15], [19].

[2]This means that every (new) literal is defined only in terms of previously known literals. The order corresponds to the order in which the solver introduced the new literals.

$$\text{IHYP-}\phi\ \frac{}{C[\mathtt{F}]}\ C \in \phi \qquad \text{IHYP-}\psi\ \frac{}{C[\mathtt{T}]}\ C \in \psi$$

$$\text{IAXI-}\phi\ \frac{}{C[\mathtt{F}]}\ C \preceq \phi, \vdash_{\mathcal{T}_U} C \qquad \text{IAXI-}\psi\ \frac{}{C[\mathtt{T}]}\ C \preceq \psi, \vdash_{\mathcal{T}_U} C$$

$$\text{IRES}\ \frac{a \vee C[I_C] \qquad \neg a \vee D[I_D]}{C \vee D[(a \vee I_C) \wedge (\neg a \vee I_D)]}\ a \preceq \phi, a \preceq \psi$$

$$\text{IRES-}\phi\ \frac{a \vee C[I_C] \qquad \neg a \vee D[ID]}{C \vee D[I_C \vee I_D]}\ a \preceq \phi, a \not\preceq \psi$$

$$\text{IRES-}\psi\ \frac{a \vee C[I_C] \qquad \neg a \vee D[I_D]}{C \vee D[I_C \wedge I_D]}\ a \not\preceq \phi, a \preceq \psi$$

Fig. 3. Interpolating proof rules

Definition 5 (Partial interpolant)**.** *Let* C *be a clause such that* $\phi \wedge \psi \rightarrow C$*. A formula* I *is a* partial interpolant *for* C *between* ϕ *and* ψ *if* $\phi \rightarrow C|_\phi \vee I$*, and* $\psi \rightarrow C|_\psi \vee \neg I$*, and* $I \preceq \phi$*, and* $I \preceq \psi$*. If* I *is a partial interpolant for* $C = \emptyset$ *between* ϕ *and* ψ*, then* I *is an interpolant between* ϕ *and* ψ.

In Figure 3, we present interpolating proof rules. In an unsatisfiability proof of $\phi \wedge \psi$, these rules annotate (in square brackets) each conclusion with a partial interpolant for the conclusion. Rules IHYP-ϕ and IHYP-ψ are used at leaf nodes that have clauses from ϕ and ψ respectively. Rules IAXI-ϕ and IAXI-ψ are used for leaves with theory-tautology clauses, whose symbols are a subset of the symbols in ϕ and ψ respectively. Note that these rules assume that the unsatisfiability proof of $\phi \wedge \psi$ is *colorable*.

Definition 6 (Colorable Proof)**.** *A proof of unsatisfiability of* $\phi \wedge \psi$ *is* colorable *if for every leaf* n_L *of the proof* $Symb(clause(n_L)) \subseteq Symb(\phi)$ *or* $Symb(clause(n_L)) \subseteq Symb(\psi)$.

In Section V, we will present an algorithm that transforms a proof into a colorable proof. Due to this assumption we can easily find corresponding partial interpolants for theory-tautology clauses, which are either `T` or `F`. For internal proof nodes, we follow Pudlák's interpolation system [19]. The annotation of the root node (with the empty clause) is the interpolant between ϕ and ψ. See [7] for a proof of correctness of the annotating proof rules.

IV. CONTROLLER SYNTHESIS

A. Overview

Following [12], we assume that synthesis problems are given as formulas of the form

$$\forall \bar{i}\ \exists \bar{c}\ \forall \bar{o}.\ \Phi(\bar{i}, \bar{c}, \bar{o}), \tag{1}$$

where $\bar{c}$ is a vector of Boolean variables and Φ is a formula over theory $\mathcal{T}_U$. Let $\bar{c} = (c_1, \ldots, c_n)$. Each c_i represents a missing if-condition in a program or a one-bit control signal in a hardware design. Witness functions for the existentially quantified variables in Eq. (1) are implementations of the missing components. Therefore, the synthesis problem is equivalent to finding such witness functions. I.e., find $(f_1(\bar{i}), \ldots, f_n(\bar{i}))$ such that $\forall \bar{i}\ \forall \bar{o}.\ \Phi(\bar{i}, (f_1(\bar{i}), \ldots, f_n(\bar{i})), \bar{o})$ holds true.

We compute the witness functions through the following steps:

(1) Expand the existential quantifier and negate the formula Φ to obtain an unsatisfiable formula ϕ (Sec. IV-B).

(2) Obtain a proof of unsatisfiability from an SMT solver.
(3) Transform the proof into a colorable, local-first proof (Sec. V).
(4) Perform n-interpolation on the transformed proof. The elements of the n-interpolant correspond to the witness functions (Sec. IV-B).

We will first introduce the notion of n-interpolation and show how it is used to find witness functions in Section IV-B. Subsequently, we will show how to transform a proof of unsatisfiability so that it is suitable for n-interpolation in Section V.

B. Finding Witness Functions through Interpolation

Jiang et al. [14] show how to compute a witness function in Eq. (1) using interpolation if $\bar{c}$ contains a single Boolean c. In this case, Eq. (1) reduces to $\forall \bar{i}\ \exists c\ \forall \bar{o}.\ \Phi(\bar{i}, c, \bar{o})$. After expanding the existential quantifier by instantiating the above formula for both Boolean values of c and renaming $\bar{o}$ in each instantiation, we obtain the equivalent formula $\forall \bar{i}\ \forall \bar{o}_{\mathtt{F}}, \bar{o}_{\mathtt{T}}.\ \Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}}) \vee \Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$. Since all the quantifiers are universal, the disjunction is valid. Therefore, its negation $\neg\Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}}) \wedge \neg\Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$ is unsatisfiable. The interpolant between the two conjuncts is the witness function for variable c.

Theorem 2. *The interpolant between $\neg\Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}})$ and $\neg\Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$ is the witness function for c. (For a proof, see [13].)*

We now extend this idea to compute witness functions when $\bar{c}$ is a vector of Booleans $(c_1, \ldots, c_n)$. Let $\mathbb{B}^n$ denote the set of vectors of length n containing Fs and Ts. For vector $w \in \mathbb{B}^n$, let w_j be the Boolean value in w at index j. Since $\bar{c}$ is a Boolean vector, we can expand the existential quantifier for $\bar{c}$ in Eq. (1) by enumerating the finitely many possible values of $\bar{c}$ to obtain $\forall \bar{i} \bigvee_{w\in\mathbb{B}^n} \forall \bar{o}\ .\ \Phi(\bar{i}, w, \bar{o})$. By dropping the quantifiers and renaming $\bar{o}$ accordingly, we obtain $\bigvee_{w\in\mathbb{B}^n} \Phi(\bar{i}, w, \bar{o}_w)$. It is valid iff Eq. (1) is valid. Let ϕ denote its negation $\bigwedge_{w\in\mathbb{B}^n} \neg\Phi(\bar{i}, w, \bar{o}_w)$, which is unsatisfiable. Let ϕ_w denote $\neg\Phi(\bar{i}, w, \bar{o}_w)$. We will call the ϕ_ws the 2^n *partitions* of ϕ. We will learn a vector of *coordinated* interpolants from an unsatisfiability proof of ϕ. These interpolant formulas will be witness functions for $\bar{c}$. Since ϕ_ws are obtained by only renaming variables, the shared symbols between any two partitions are equal.

Definition 7 (Global and Local Symbols)**.** *Symbols in the set $G = \bigcap_{w\in\mathbb{B}^n} Symb(\phi_w)$ are called* global *symbols. All other symbols are called* local *(w.r.t. the one partition in which they occur).*

Let $\overline{I}$ be a vector of formulas $(I_1, \ldots, I_n)$. Let $\oplus$ be the exclusive-or (xor) operator. For a word $w \in \mathbb{B}^n$, let $\overline{I}' = \overline{I} \oplus w$ if for each $j \in 1..n$, $I'_j = I_j \oplus w_j$. Let $\bigvee \overline{I}$ be short for $I_1 \vee \cdots \vee I_n$. Let $C|_w = C|_{\phi_w}$. The following definition generalizes the notion of interpolant and partial interpolant from two formulas to 2^n formulas.

Definition 8 (n-Partial Interpolant)**.** *Let C be a clause such that $(\bigwedge_{w\in\mathbb{B}^n} \phi_w) \rightarrow C$. An n-partial interpolant $\overline{I}$ for C w.r.t. the ϕ_ws is a vector of formulas with length n such that $\forall w \in \mathbb{B}^n.\ \phi_w \rightarrow (C|_w \vee \bigvee(\overline{I} \oplus w))$ and $\overline{I} \preceq G$. If $C = \emptyset$ then $\overline{I}$ is an* n-interpolant *w.r.t. the ϕ_ws.*

$$\textsc{mHyp}\ \frac{}{C[w]}\ C \in \phi_w \qquad \textsc{mAxi}\ \frac{}{C[w]}\ C \preceq \phi_w$$

$$\textsc{mRes}\ \frac{a \vee C[w] \qquad \neg a \vee D[w]}{C \vee D[w]}\ w \in \mathbb{B}^n, a \vee C \vee D \preceq \phi_w$$

$$\textsc{mRes-G}\ \frac{a \vee C[\overline{I}^C] \qquad \neg a \vee D[\overline{I}^D]}{\begin{array}{l} C \vee D\ \ [(\ (a \vee I_1^C) \wedge (\neg a \vee I_1^D), \\ \qquad \ldots, \\ \qquad (a \vee I_n^C) \wedge (\neg a \vee I_n^D)\ )] \end{array}}\ a \preceq G$$

Fig. 4. n-Interpolating proof rules for an unsatisfiable $\phi = \bigwedge_{w\in\mathbb{B}^n} \phi_w$. These rules can only annotate proofs that are colorable and local-first.

Theorem 3. *An n-interpolant w.r.t. the ϕ_ws constitutes witness functions for the variables in $\bar{c}$. (For a proof see [13].)*

C. Computing n-interpolants

In Figure 4, we present the proof rules for n-interpolants. These rules annotate each conclusion of a proof step with an n-partial interpolant for the conclusion w.r.t. the ϕ_ws. These annotation rules require two properties of the proof. First, it needs to be colorable.[3] Second, it needs to be local-first.

Definition 9 (Local-first Proof)**.** *A proof of unsatisfiability is* local-first*, if for every resolution node with a local pivot both its premises are derived from the same partition.*

The rule MHYP annotates the derived clause C with w if C appears in partition ϕ_w. Similarly, the rule MAXI annotates theory-tautology clause C with w if $C \preceq \phi_w$. Rules MRES and MRES-G annotate resolution steps. MRES-G is only applicable if the pivot is global and follows Pudlák's interpolation system n times. MRES is only applicable if both premises are annotated with the same n-partial interpolant and this n-partial interpolant is an element of $\mathbb{B}^n$. Due to the local-first assumption on proofs, these rules will always be able to annotate a proof.

Theorem 4. *Annotations in the rules in Figure 4 are n-partial interpolants for the respective conclusions w.r.t. the ϕ_ws. (For a proof see [13].)*

Since the n-interpolant is always quantifier free, we can easily convert it into an implementation. To create a circuit for one element of the n-interpolant, we create, for every resolution node with a global pivot, a multiplexer that has the pivot at its selector input. The other inputs connect to the outputs of the multiplexers corresponding to the child nodes. For leaf nodes and resolution nodes with local pivots, we use the constants $\mathtt{T}, \mathtt{F}$, depending on which partition the node belongs to. The output of the multiplexer corresponding to the root node is the final witness function. Note that, unless we apply logical simplifications, the circuits for all witness functions all have the same multiplexer tree and differ only in the constants at the leaves of this tree.

Also note that due to the local-first property, all nodes that are derived from a single partition are annotated with the same n-partial interpolant. Thus, we can disregard such

[3] We extend Def. 6 from two formulas to 2^n partitions in the obvious way.

local sub-trees, by iteratively converting nodes that have only descendants from one partition into leaves. This does not affect the outcome of the interpolation procedure.

The local-first property is actually needed to correctly compute witness functions using Pudlák's interpolation system. In [13], we illustrate this observation with an example. Also note that McMillan's interpolation [18] system does not produce correct witness functions even with the local-first property.

V. Algorithms for Proof Transformation

Our interpolation procedure requires proofs to be colorable and local-first. These properties are not guaranteed by efficient modern SMT solvers. In this section we will show how to transform a proof conforming to Def. 2 into one that is colorable and local-first. Our proof transformation works in three steps. First, we will remove any non-colorable literals from the proof. Second, we will split any non-colorable theory-tautology clauses. This gives us a colorable proof. In the third step, we will reorder resolution steps to obtain the local-first property [7]. For ease of presentation, we will assume that the proof is a tree (instead of a DAG). The method extends to proofs in DAG form.

A. Removing Non-Colorable Literals

Definition 10 (Colorable and Non-Colorable Literals)**.** *A literal a is* colorable *with respect to a partition ϕ_w (w-*colorable*) iff $a \preceq \phi_w$. A literal that is not w-colorable for any partition w is called* non-colorable*.*

Note that global literals are w-colorable for every w. By definition, the formula ϕ is free of non-colorable literals (equalities and predicate instances). Thus, the only way through which non-colorable literals can be introduced into the proof are theory-tautology clauses.

We search the proof for a theory-tautology clause that introduces a non-colorable literal a and has only colorable literals as implying literals. We call this proof node the *defining node* n_d. At least one such leaf must exist. We remove this non-colorable literal from the proof as follows. Starting from n_d, we traverse the proof towards the root, until we find a node, which we call *resolving node* n_r, whose clause does not contain the literal a any more. Since the root node does not contain any literals, such a node always exists. Let n_a and $n_{\neg a}$ be the premises of n_r, respectively, depending on which phase of literal a their clause contains. From $n_{\neg a}$, we traverse the proof towards the leaves along nodes that contain the literal $\neg a$. Note that any leaf that we reach in this way must be labeled with a theory-tautology clause, as clauses from ϕ cannot contain the non-colorable literal $\neg a$. Note that $\neg a$ is among the implying literals of such a leaf node's clause. I.e., such nodes *use* the literal to imply another one. We will therefore call such a node a *using node* n_u. We update $clause(n_u)$, by removing $\neg a$ and adding the implying literals of $clause(n_d)$ instead.

It is easy to see that this does not affect $clause(n_u)$'s property of being a theory-tautology clause. Suppose $clause(n_d)$ is $(\neg x_1 \vee \ldots \vee \neg x_k \vee a)$. Then $\bigwedge_{i=1}^{k} x_i \rightarrow a$. By reversing the implication we obtain $\neg a \rightarrow \bigvee_{i=1}^{k} \neg x_i$. Therefore, replacing $\neg a$ with the disjunction of the implying literals of $clause(n_d)$ in $clause(n_u)$ is sound.

To keep the proof internally consistent, we have to do the same replacement on all the nodes on the path between n_u and n_r. The node n_r itself is not changed, as $clause(n_r)$ does not contain the non-colorable literal $(\neg)a$ any more. I.e., the last node that is updated is the node $n_{\neg a}$.

Now we have to distinguish two cases. The first case is that node n_a still contains all of the implying literals of n_d. In this case, $clause(n_r) = clause(n'_{\neg a})$, where $n'_{\neg a}$ is the updated node $n_{\neg a}$. Thus, we use $n'_{\neg a}$ instead of n_r in n_r's parent node. In the second case, some of the implying literals of $clause(n_d)$ have already been resolved on the path from n_d to n_r. In that case $clause(n'_{\neg a})$ contains literals that do not occur in $clause(n_r)$. Let x_l be one such literal. We search the path from n_d to n_r for the node that uses x_l as a pivot. Its premise that is not on the path from n_d to n_r contains $\neg x_l$. We use this node and the node $n'_{\neg a}$ as premises for a new resolution node with x_l as pivot. Note that this resolution may introduce more literals that do not appear in $clause(n_r)$ any more. However, just as with x_l, any such literal must have been resolved somewhere on the path between n_d and n_r. Thus, we repeat this procedure, replicating the resolution steps that took place between n_d and n_r, until we get a node whose clause is identical to $clause(n_r)$. This node can then be used instead of n_r in n_r's parent node. Finally, we remove all nodes that are now unreachable from the proof.

Example 1. An illustrative example of this procedure is shown in Figure 5.

We repeat this procedure for all leaves with a non-colorable implied literal and (all) colorable implying literals. Note that one application of this procedure may convert a node where a non-colorable literal was implied by at least one other non-colorable literal into a node where the implied non-colorable literal is now implied only by colorable literals. Nevertheless this procedure terminates, as the number of leaves with non-colorable implied literals decreases with every iteration. Each iteration removes (at least) one such leaf from the proof and no new leaves are introduced.

Theorem 5. *Upon termination of this procedure, the proof does not contain any non-colorable literals.*

B. Splitting Non-Colorable Theory-Tautology Clauses

After removing all non-colorable literals, the proof may still contain non-colorable theory-tautology clauses, i.e., theory-tautology clauses that contain local literals from more than one partition. We split such leaves into several new theory-tautology clauses, each containing only w-colorable literals, and, via resolution, obtain a (now internal) node with the same clause as the original non-colorable theory-tautology clause. Note that internal nodes with non-colorable clauses are not a problem for our interpolation procedure, but leaves with non-colorable clauses are. We will show how to split a non-colorable theory-tautology clause with an implied equality

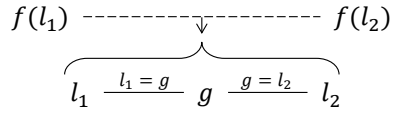
$$\textsc{Res}\,\frac{\textsc{Res}\,\dfrac{n_1 : (l_1 = z_g \vee x_g = y_g)\;\; n_d : (l_1 \neq z_g \vee z_g \neq l_2 \vee l_1 = l_2)}{n_a : (x_g = y_g \vee z_g \neq l_2 \vee l_1 = l_2)} \quad \textsc{Res}\,\dfrac{n_u : (l_1 \neq l_2 \vee f(l_1) = f(l_2))\;\; n_3 : (f(l_1) \neq f(l_2) \vee u_g \neq v_g)}{n_{\neg a} : (l_1 \neq l_2 \vee u_g \neq v_g)}}{n_r : (x_g = y_g \vee z_g \neq l_2 \vee u_g \neq v_g)}$$

(a) Proof before removing non-colorable literal $l_1 = l_2$.

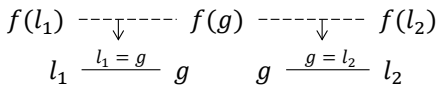
$$\textsc{Res}\,\frac{n_1 : (l_1 = z_g \vee x_g = y_g) \quad \textsc{Res}\,\dfrac{n_3 : (f(l_1) \neq f(l_2) \vee u_g \neq v_g)\;\; n'_u : (l_1 \neq z_g \vee z_g \neq l_2 \vee f(l_1) = f(l_2))}{n'_{\neg a} : (l_1 \neq z_g \vee z_g \neq l_2 \vee u_g \neq v_g)}}{n''_{\neg a} : (x_g = y_g \vee z_g \neq l_2 \vee u_g \neq v_g)}$$

(b) Proof after removing non-colorable literal $l_1 = l_2$.

Fig. 5. *Removing a non-colorable literal.* Assume that term indices indicate the number of the partition the term belongs to. Index g is used for global terms. This example shows how the non-colorable literal $l_1 = l_2$, introduced in node n_d, is removed from the proof by replacing its negative occurrences with the (colorable) defining literals $(l_1 \neq z_g \vee z_g \neq l_2)$. Note that in the original proof $l_1 \neq z_g$ is already resolved on the path from n_d to n_r using node n_1. This resolution step is replicated in the transformed proof by making a resolution step with nodes $n'_{\neg a}$ and n_1. Since the literal $x_g = y_g$ introduced into $n''_{\neg a}$ also occurs in the original n_r, and also the second defining literal $z_g \neq l_2$ occurs in n_r, no further resolution steps are necessary. The conclusions of n_d and $n''_{\neg a}$ are identical and $n''_{\neg a}$ can be used instead of n_r in n_r's parent.



(a) Non-Colorable Transitivity-Congruence Chain for $(f(l_1) \rightsquigarrow f(l_2))$



(b) Colorable Transitivity-Congruence Chain for $(f(l_1) \rightsquigarrow f(l_2))$

Fig. 6. Splitting a non-colorable transitivity-congruence chain by introducing global intermediate terms.

literal. This procedure can be trivially extended to implied literals that are uninterpreted predicate instances.

Using the implying literals of the theory-tautology clause (converted to their positive phase), we create a *congruence graph* (cf. Def. 3). Since the implying literals and the implied literal form a theory tautology, this congruence graph is guaranteed to contain a path between the the two terms equated by the implied literal. We use breadth-first search to find the shortest such transitivity-congruence chain (Def. 4).[4] The chain will be the basis for splitting the non-colorable theory tautology. First, we need to make all edges in the chain colorable. A *colorable edge* is an edge for which there is a w such that all the edge's literals are w-colorable. Edges with an equality justification already are colorable, as we assumed that no non-colorable literals occur in the theory-tautology clause. Edges with congruence justifications, however, may still be non-colorable. I.e., the two terms they connect might belong to different partitions, and/or some of the paths that prove equality for the function parameters might span over more than one partition. Fuchs et al. [8] have shown how to recursively make all edges in a chain colorable by introducing global intermediate terms for non-colorable edges. We will illustrate this procedure with a simple example, and refer to [8] for details.

Example 2. Suppose we have the two local terms $f(l_1)$ and $f(l_2)$, where l_1, l_2 are from two different partitions, and a global term g. (See Fig. 6.) A possible (non-colorable) congruence justification for $f(l_1) = f(l_2)$ could be given as $(l_1 = g, g = l_2)$. The edge between $f(l_1)$ and $f(l_2)$ is now split into two (colorable) parts: $f(l_1) = f(g)$, with justification $l_1 = g$, and $f(g) = f(l_2)$, with justification $g = l_2$. Note that $f(g)$ is a new term that (possibly) did not appear in the congruence graph before. Since we assumed that there are no non-colorable equality justifications in our graph, such a global intermediate term must always exist. It should be clear how to extend this procedure to n-ary functions.

Note that in a colorable chain, every edge either connects two terms of the same partition, or a global term and a local term. In other words, terms from different partitions are separated by at least one global term between them. We now divide the whole chain into (overlapping) segments, so that each segment uses only w-colorable symbols. The global terms that separate symbols with different colors are part of both segments.[5] Let's assume for the moment that the chain starts and ends with a global term. We will show how to deal with local terms at the beginning/end of the chain later. For ease of presentation, also assume that the chain consists of only two segments. An extension to chains with more segments can be done by recursion. We take the first segment of the chain (from start to the global term that is at the border to the next segment), plus a new "shortcut" literal that states equality between the last term of the first segment and the last term of the entire chain, and use them as implying literals for a new theory-tautology clause. The implied literal of this tautology will be an equality between the first and the last term of the entire chain. Next, we create a theory tautology with the literals of the second segment of the chain. Note that the implied literal of this theory-tautology clause (which occurs in positive phase) is the same as the shortcut literal used in the theory-tautology clause corresponding to the first segment. There, however, it occurred in negative phase. Thus, we can use this literal for resolution between the two clauses. We obtain a node that has all the literals of the entire chain as implying literals, and an equality between start term and end term of the chain as the implied literal. I.e., this new internal node has the same conclusion as the non-colorable theory-tautology clause from which we started.

In case the start/end of the chain is not a local term, we

[4]Note that these graphs are usually relatively small.

[5]If there is more than one consecutive global term, we arbitrarily choose the last one.

$$\textsc{Res}\,\frac{\textsc{Res}\,\dfrac{n_1 : [c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq k_g \vee c_g = k_g] \quad n_2 : [f_g \neq h_3 \vee h_3 \neq k_g \vee f_g = k_g]}{n_3 : [c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee c_g = k_g]} \quad n_4 : [a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1]}{n_5 : [a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1]}$$

Fig. 7. *Splitting theory tautology clauses.* Suppose we have created the transitivity-congruence chain $(a_1 \rightsquigarrow b_1 \rightsquigarrow c_g \rightsquigarrow d_2 \rightsquigarrow e_2 \rightsquigarrow f_g \rightsquigarrow h_3 \rightsquigarrow k_g \rightsquigarrow l_1)$ from a theory-tautology clause, where all the edges are colorable. The number in the index indicates the partition of the respective term, with g being used for global terms. First, we consider only the part from the first to the last global term (c_g and k_g, respectively). We "split" this sub-chain into the chains $(c_g \rightsquigarrow d_2 \rightsquigarrow e_2 \rightsquigarrow f_g \rightsquigarrow k_g)$ and $(f_g \rightsquigarrow h_3 \rightsquigarrow k_g)$ and convert them into (colorable) theory tautology clauses (nodes n_1 and n_2, respectively). By resolution we obtain n_3. Now, we create the tautology in node n_4, which corresponds to all links of the original chain which we have not dealt with already, and a "shortcut" over the part we have already considered: $(a_1 \rightsquigarrow b_1 \rightsquigarrow c_g \rightsquigarrow k_g \rightsquigarrow l_1)$. Note that this is also a colorable theory-tautology clause. By resolution over n_3 and n_4 we obtain n_5, whose clause is identical to the theory-tautology clause from which we started.

first deal with the sub-chain from the first to the last global term, as described above. Note that if both start and end of the chain are local terms, they have to belong to the same partition, because otherwise the implied literal would be non-colorable. We create a theory-tautology clause with the local literals from the start/end of the chain, and one shortcut literal that equates the first and last global term. This literal can be used for resolution with the implied literal of the node obtained in the previous step.

In summary, this procedure replaces all leaves that have non-colorable theory-tautology clauses with subtrees whose leaves are all colorable theory-tautology clauses, and whose root is labeled with the same clause as the original non-colorable leaf.

Example 3. Fig. 7 shows how to split the non-colorable theory-tautology clause $(a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1)$.

Theorem 6. *After applying the above procedure to all leaves with non-colorable theory-tautology clauses, the proof is colorable.*

C. Obtaining a local-first proof

To obtain a local-first proof, we traverse the proof in topological order. Each time we encounter a resolution step that has a global pivot and we have seen local pivots among its ancestors then we apply one of the two transformation rules presented in Figure 8 depending on the matching pattern. These two transformation rules are the standard pivot reordering rules from [7]. Note that these rules assume that the proof is redundancy free, which can be achieved by the algorithms presented in [10]. After repeated application of these transformation rules, we can move the resolutions with local pivots towards the leaves of the proof until we don't have any global pivot among its descendants.

Theorem 7. *After exhaustive application of this transformation, we obtain a colorable, local-first proof.*

VI. Experimental Results

We have implemented a prototype to evaluate our interpolation-based synthesis approach. We read the formula Φ corresponding to our synthesis problem (Eq. 1) from a file in SMT-LIB format [1]. As a first step, our tool performs several transformations on the input formula (reduction of arrays to uninterpreted functions [4], expansion of the existential quantifier to obtain the partitions, renaming of $\bar{o}$-variables in each partition, negation to obtain ϕ), before giving it to the VERIT solver. Second, we apply the proof transformations described in Section V to the proof we obtain from VERIT. Third, we compute the witness functions as the n-interpolants w.r.t. the partitions of ϕ.

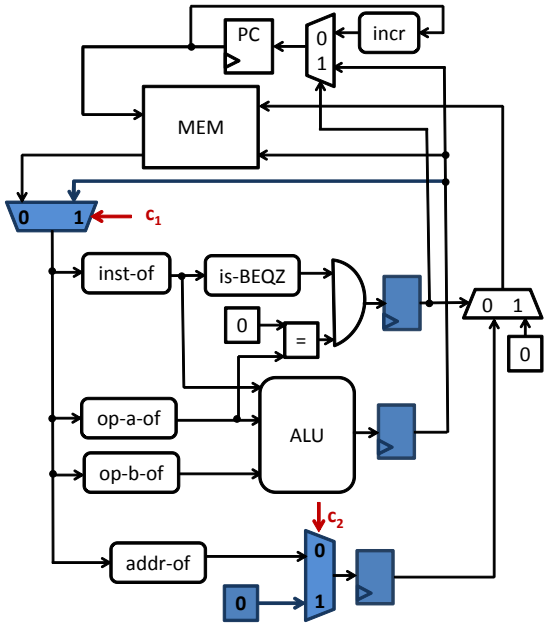


Fig. 9. A simple microprocessor with a 2-stage pipeline.

We have checked all results using Z3 [6], by showing that $\neg\Phi(\bar{i}, (f_1(\bar{i}), \ldots, f_n(\bar{i})), \bar{o})$ is unsatisfiable.

We used our tool on several small examples and also tried one non-trivial example which we explain in more detail. In Fig. 9 we show a simple (fictitious) microprocessor with a 2-stage pipeline. MEM represents the main memory. We assume that the value at address 0 is hardwired to 0. I.e., reading from address 0 always yields value 0. The blocks inst-of, op-a-of, op-b-of, and addr-of represent combinational functions that decode a memory word. The block incr increments the program counter (PC). The block is-BEQZ is a predicate that checks whether an instruction is a branch instruction. The design has two pipeline-related control signals for which we would like to synthesize an implementation. Signal c_1 causes a value in the pipeline to be forwarded and signal c_2 squashes the instruction that is currently decoded and executed in the first pipeline stage. This might be necessary due to speculative execution based on a "branch-not-taken assumption". The implementation of these control signals is not as simple as it might seem at first glance. For example, the seemingly trivial solution of setting $c_1 = \mathtt{T}$ whenever PC equals the address register is not correct. For example, if PC $= 0$, forwarding

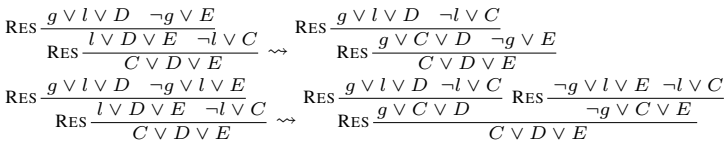


Fig. 8. If a local pivot l occurs after a global pivot g in a proof then we can rewrite the proof using one of the above transformation rules. After the transformation, the proof first resolves l then g.

TABLE I

Experimental results. Columns: (1) Name; (2) Number of control signals; (3) Total synthesis time including checking the results; (4) Number of leaves with theory-tautology clauses that define a new non-colorable literal (Number of such leaves at the start of the cleaning procedure + Number of leaves introduced (and subsequently removed) by the procedure); (5) Number of leaves to be split because they contain literals from more than one partition. (Number after "/" is total number of leaves in proof at beginning of split procedure; (6) Time to reorder the proof to be local-first; (7) Number of nodes in proof from VERIT / Size of the transformed proof for interpolation (local sub-trees have been converted to leaves).

Name	Ctrl	time [s]	# leaves to clean	# leaves to split	Reorder-Time [ms]	Proof size
const	2	0.6	0	0 / 6	42	19 / 1
illu02	2	1.1	1	1 / 65	83	205 / 12
illu03	3	5.0	8	8 / 138	487	467 / 22
illu04	4	8.0	3	3 / 242	532	951 / 75
illu05	5	12.8	10	10 / 413	589	1588 / 78
illu06	6	237.0	9	9 / 1093	1820	4691 / 370
illu07	7	150.0	14	14 / 1443	2860	6824 / 555
illu08	8	1270.0	20	20 / 3450	4980	17524 / 1023
pipe	1	1.6	6 + 6	3 / 70	129	285 / 22
proc	2	28.1	3 + 3	61 / 1014	1770	5221 / 1042

should not be done.[6] By taking out the blue parts in Fig. 9 we obtain the non-pipelined reference implementation which we used to formulate a Burch-Dill-style equivalence criterion [5]. The resulting formula was used as a specification for synthesis.

Table I summarizes our experimental results. The benchmark "const" is a simple example with 2 control signals that allows for constants as valid solutions. "illu02" is the example presented in Section II; "illu03" to "illu08" are scaled-up versions of "illu02", with increased numbers of inputs and control signals. "pipe" is the simple pipeline example that was used in [12]. "proc" is the pipelined processor shown in Fig. 9 and described above. All experiments were performed on an Intel Nehalem CPU with 3.4 GHz.

Note that using our new method we have reduced the synthesis time of "pipe" from 14 hours [12] to 1.6 seconds. As a second comparison, we tried to reduce the (quantified) input formula of "proc" to a QBF problem (using the transformations outlined in [12]) and run DEPQBF [17] on it. After approximately one hour, DEPQBF exhausted all 192 GB of main memory and terminated without a result.

VII. CONCLUSION

Hofferek and Bloem [12] have shown that uninterpreted functions are an efficient way to abstract away unnecessary details in controller synthesis problems. By using interpolation in $\mathcal{T}_U$, we avoid the costly reduction to propositional logic, thus unleashing the full potential of the approach presented in [12]. Furthermore, by introducing the concept of n-interpolation, we also avoid the iterative construction which requires several calls to the SMT solver and back-substitution. The n-interpolation approach improves synthesis times by several orders of magnitude, compared to previous methods [12], rendering it applicable to real-world problems, such as pipelined microprocessors. We have also shown that a naive transformation to QBF is not a feasible option.

[6] We actually made this mistake while trying to create and model-check a manual implementation for the control signals, and it took some time to locate and understand the problem.

REFERENCES

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories*, 2010.

[2] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *DATE*, 2007.

[3] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In *CADE*, 2009.

[4] A. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*. Springer, 1994. LNCS 818.

[6] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[7] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI*, 2010.

[8] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*. ACM, 2011.

[10] A. Gupta. Improved single pass algorithms for resolution proof reduction. In *ATVA*. Springer, 2012.

[11] K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In *POPL*. ACM, 2012.

[12] G. Hofferek and R. Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *MemoCODE*. IEEE, 2011.

[13] G. Hofferek, A. Gupta, B. Könighofer, J.-H. R. Jiang, and R. Bloem. Synthesizing multiple boolean functions using interpolation on a single proof, 2013. Full version with appendix available at arXiv.org:1308.4767.

[14] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung. Interpolating functions from large Boolean relations. In *ICCAD*, 2009.

[15] L. Kovács and A. Voronkov. Interpolation and symbol elimination. In *CADE*. Springer, 2009.

[16] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*. ACM, 2010.

[17] F. Lonsing and A. Biere. Integrating dependency schemes in search-based QBF solvers. In *SAT 2010*, 2010.

[18] K. L. McMillan. An interpolating theorem prover. *TCS*, 345(1), 2005.

[19] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 1997.

[20] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*. ACM, 2007.

[21] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*. ACM, 2005.

[22] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*. ACM, 2006.

[23] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*. ACM, 2011.

[24] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*. ACM, 2010.

[25] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

Quantifier Elimination via Clause Redundancy

Eugene Goldberg and Panagiotis Manolios
Northeastern University, USA, {eigold,pete}@ccs.neu.edu

***Abstract*—We consider the problem of existential quantifier elimination for Boolean formulas in conjunctive normal form. Recently we presented a new method for solving this problem based on the machinery of Dependency sequents (D-sequents). The essence of this method is to add to the quantified formula implied clauses until all the clauses with quantified variables become redundant. A D-sequent is a record of the fact that a set of quantified variables is redundant in some subspace. In this paper, we introduce a quantifier elimination algorithm based on a new type of D-sequents called *clause D-sequents* that express redundancy of clauses rather than variables. Clause D-sequents significantly extend our ability to introduce and algorithmically exploit redundancy, as our experimental results show.**

I. Introduction

In this paper, we consider elimination of quantifiers from formulas of the form $\exists X[F]$ where F is a Boolean formula in conjunctive normal form (CNF). We will refer to such formulas as $\exists$CNF. The **Quantifier Elimination (QE) problem**, is to find a quantifier-free CNF formula G such that $G \equiv \exists X[F]$. The equivalence '$\equiv$' is semantic. That is for every complete assignment $\boldsymbol{s}$ to the non-quantified variables of F, the logical value of $G_{\boldsymbol{s}}$ is equal to that of $\exists X[F_{\boldsymbol{s}}]$. Here $F_{\boldsymbol{s}}$ and $G_{\boldsymbol{s}}$ are formulas F and G under assignment $\boldsymbol{s}$.

The motivation for studying the QE problem is twofold. First, a QE algorithm can be used for solving many verification problems e.g. computing reachable states [5], [17]. Second, the methods developed for QE may come handy for other problems. For example, the machinery of Dependency sequents (D-sequents) [9], [10] that we continue developing in this paper can be used for SAT-solving [8].

In [9], [10], we developed a QE algorithm called DDS (Derivation of D-Sequents) based on the following two ideas. The first idea is that adding resolvent clauses to formula F eventually makes the clauses containing a variable of X (we will call them ***X*-clauses**) redundant. Let H denote formula $F \wedge G$ where G is the set of added resolvent clauses. For the sake of convenience, since a CNF formula can be considered as a set of clauses, we will use logical and set-based notation interchangeably. For example, formula $F \wedge G$ can also be written as $F \cup G$. The redundancy of X-clauses in $\exists X[H]$ means that $\exists X[H] \equiv \exists X[H \setminus H^X]$ where H^X is the set of X-clauses of H. Since $H \setminus H^X$ does not depend on X, the quantifiers can be dropped. So the set of clauses $H \setminus H^X$ is a quantifier-free formula equivalent to $\exists X[H]$ and so to $\exists X[F]$. The second idea is to use a divide-and-conquer strategy. DDS proves redundancy of X-clauses in subspaces and then merges the results of different branches.

A successful implementation of DDS became possible only due to development of the machinery of D-sequents that was the main contribution of [9], [10]. A D-sequent is a record of the form $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ where $\boldsymbol{q}$ is a partial assignment to variables of F and $Z \subseteq X$. This D-sequent says that the variables of Z are redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. The redundancy of variables of Z means redundancy of all X-clauses containing a variable of Z. For the sake of brevity, in the following exposition we use the same symbol F to denote the initial and the *current* CNF formula consisting of the initial clauses and *resolvents*. So symbol F used in the D-sequent above is the current CNF formula.

DDS keeps adding resolvent clauses to formula F until D-sequent $(\exists X[F], \emptyset) \rightarrow X$ is derived stating that the variables of X are redundant in formula $\exists X[F]$ globally. (This means that $F \setminus F^X$ is a solution to the QE problem.) The derivation of such D-sequent is achieved by generation of atomic D-sequents and using a resolution-like operation *join*. An atomic D-sequent is derived when redundancy of a variable of X in a subspace can be trivially proved. Operation *join* is applied to D-sequents $(\exists X[F], \boldsymbol{q'}) \rightarrow Z$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow Z$ where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ contain opposite assignments to a variable v of F. The result of *join* is a new D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ where $\boldsymbol{q}$ consists of all assignments of $\boldsymbol{q'}$,$\boldsymbol{q''}$ but those to variable v.

The main contribution of this paper is the development of the machinery of a new type of D-sequents called **clause D-sequents**. A clause D-sequent is a record of the form $(\exists X[F], \boldsymbol{q}) \rightarrow R$ where $R \subseteq F^X$. It states that the X-clauses of R are redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. Clause D-sequents can be used to express redundancy of *any* subset of X-clauses while D-sequents of [9] can do this only for *some* subsets of X-clauses. Namely, a D-sequent of [9] can express redundancy of a set $R \subseteq F^X$ *only* if R is the set of *all* X-clauses containing variables from a set Z. (To distinguish new and old D-sequents we will refer to the latter as D-sequents based on variable redundancy.) For instance, a D-sequent based on variable redundancy cannot express the fact that a single X-clause C is redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. Similarly to D-sequents based on variable redundancy, the machinery of clause D-sequents is based on a) derivation of atomic clause D-sequents, b) a resolution-like operation *join* and c) removing redundant X-clauses from the formula to guarantee the composability of D-sequents. The latter means that proving redundancy of sets of clauses R' and R'' independently implies that the set $R' \cup R''$ is also redundant.

To show the advantages of clause D-sequents we describe a new QE algorithm called ***DCDS*** (Derivation of Clause D-Sequents). Development of $DCDS$ is another contribution of this paper. $DCDS$ can be viewed as an adaptation of DDS to clause D-sequents. However, this adaptation is far from being trivial because clause D-sequents have new features that D-sequents based on variable redundancy do not. Using clause D-sequents is beneficial for at least two reasons. First, $DCDS$ has much more flexibility than DDS in proving redundancy of X-clauses. Proving that a variable $v \in X$ is redundant in $\exists X[F]$

ISBN 978-0-9835678-3-7/13.

in subspace $\boldsymbol{q}$ by DDS requires proving redundancy of all clauses of F with variable v *at the same time*. $DCDS$ has no such restriction. Some clauses with variable v may be proved redundant much later than the others. Second, the size of clause D-sequents is in general *smaller* than that of D-sequents based on variable redundancy. (The size of D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ is the number of variables assigned in $\boldsymbol{q}$.) The reason is that proving redundancy of a clause is easier than that of a variable. This facilitates pruning the search space like derivation of shorter clauses does in SAT-solving.

This paper is structured as follows. In Section II, we give basic definitions. Simple cases of clause redundancy are discussed in Section III. Clause D-sequents are introduced in Section IV. Section V describes a new QE algorithm called $DCDS$. In Section VI, we explain $DCDS$ by a simple example. Experimental results are given in Section VII. Background is discussed in Section VIII, and conclusions are presented in Section IX.

II. Basic Definitions

Definition 1: An $\boldsymbol{\exists CNF\ \ formula}$ is a quantified CNF formula of the form $\exists X[F]$ where F is a CNF formula, and X is a set of Boolean variables. If we do not explicitly specify whether we are referring to CNF or $\exists$CNF formulas, when we write "formula" we mean either a CNF or $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment, F be a CNF formula, and C be a clause. $Vars(\boldsymbol{q})$ denotes the variables assigned in $\boldsymbol{q}$; $Vars(F)$ denotes the set of variables of F; $Vars(C)$ denotes the variables of C; and $Vars(\exists X[F]) = Vars(F) \setminus X$.

We consider *true* and *false* as a special kind of clauses. A non-empty clause C becomes *true* when it is satisfied by an assignment $\boldsymbol{q}$ i.e. when a literal of C is set to *true* by $\boldsymbol{q}$. A clause C becomes *false* when it is falsified by $\boldsymbol{q}$ i.e. when all the literals of C are set to *false* by $\boldsymbol{q}$.

Definition 2: Let C be a clause, H be a formula, and $\boldsymbol{q}$ be an assignment such that $Vars(\boldsymbol{q}) \subseteq Vars(H)$. Denote by $C_{\boldsymbol{q}}$ the clause equal to *true* if C is satisfied by $\boldsymbol{q}$; otherwise $C_{\boldsymbol{q}}$ is the clause obtained from C by removing all literals falsified by $\boldsymbol{q}$. $H_{\boldsymbol{q}}$ denotes the formula obtained from H by replacing every clause C of H with $C_{\boldsymbol{q}}$. In the context of this paper, it is convenient to assume that clause $C_{\boldsymbol{q}}$ equal to *true* remains in $H_{\boldsymbol{q}}$ rather than being removed from $H_{\boldsymbol{q}}$. We treat such a clause as *redundant* in $H_{\boldsymbol{q}}$ (see Proposition 1).

Definition 3: Let G, H be formulas. We say that G, H are **equivalent**, written $G \equiv H$, if for all assignments, $\boldsymbol{y}$, such that $Vars(\boldsymbol{y}) \supseteq (Vars(G) \cup Vars(H))$, we have $G_{\boldsymbol{y}} = H_{\boldsymbol{y}}$ (notice that $G_{\boldsymbol{y}}$ and $H_{\boldsymbol{y}}$ have no free variables, so by $G_{\boldsymbol{y}} = H_{\boldsymbol{y}}$ we mean semantic equivalence). Observe that if $Vars(\boldsymbol{q}) \subseteq Vars(\exists X[F])$, then $(\exists X[F])_{\boldsymbol{q}} \equiv \exists X[F_{\boldsymbol{q}}]$.

Definition 4: The **Quantifier Elimination (QE) problem** for $\exists$CNF formula $\exists X[F]$ consists of finding a CNF formula G such that $G \equiv \exists X[F]$.

Definition 5: Let Z be a set of variables. A clause C of F is called a $\boldsymbol{Z}$**-clause** if $Vars(C) \cap Z \neq \emptyset$. We denote by $\boldsymbol{F^Z}$ the set of all Z-clauses of F.

Definition 6: Let X be a set of Boolean variables, F be a CNF formula and R be a subset of X-clauses of F. The clauses of R are **redundant** in CNF formula F if $F \equiv (F \setminus R)$. The clauses of R are **redundant** in $\exists$CNF formula $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus R]$. Note that $F \equiv (F \setminus R)$ implies $\exists X[F] \equiv \exists X[F \setminus R]$ but the opposite is not true.

III. Simple Cases of Clause Redundancy

In this section, we describe three situations where clause redundancy can be trivially proved (Propositions 1, 2, 3).

Proposition 1: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$ satisfying an X-clause C of F. Then $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

Due to lack of space we omit proofs. Note that proofs of non-trivial propositions of this paper are similar to those of [10] given for D-sequents based on variable redundancy.

Proposition 2: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let C, C' be two clauses of F. Let C be falsified by $\boldsymbol{q}$ and C' be an X-clause. Then $C'_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

To formulate Proposition 3 below, we need to introduce a few definitions.

Definition 7: Let C' and C'' be clauses having opposite literals of exactly one variable $v \in Vars(C') \cap Vars(C'')$. The clause C consisting of all literals of C' and C'' but those of v is called the **resolvent** of C',C'' on v. Clause C is said to be obtained by **resolution** on v. Clauses C',C'' are called **resolvable** on v.

Definition 8: A clause C of a CNF formula F is called **blocked** at variable v, if no clause of F is resolvable with C on v. The notion of blocked clauses was introduced in [15].

Proposition 3: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let C be an X-clause of F not satisfied by $\boldsymbol{q}$ and $v \in X$ be a variable of C such that $v \notin Vars(\boldsymbol{q})$. Let clause $C_{\boldsymbol{q}}$ be blocked at v in $F_{\boldsymbol{q}}$. Then $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

IV. Dependency Sequents Based on Clause Redundancy

In this section, we define a new kind of dependency sequents (D-sequents) called clause D-sequents. In contrast to D-sequents of [9], clause D-sequents are based on the notion of clause redundancy. We describe operation *join* producing a new clause D-sequent from existing ones and introduce the notion of composable clause D-sequents.

A. Definition of D-sequents

Definition 9: Let $\exists X[F]$ be an $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment to $Vars(F)$ and R be a subset of X-clauses of F. A clause dependency sequent (**clause D-sequent**) has the form $(\exists X[F], \boldsymbol{q}) \rightarrow R$. It states that the clauses of $R_{\boldsymbol{q}}$ are redundant in $\exists X[F_{\boldsymbol{q}}]$.

From now on we will refer to clause D-sequents as *just* D-sequents unless we want to contrast clause D-sequents with those based on variable redundancy.

Example 1: Consider an $\exists$CNF formula $\exists X[F]$ where $F = C_1 \wedge C_2$, $C_1 = x \vee y_1$ and $C_2 = \overline{x} \vee y_2$ and $X = \{x\}$. Let

$\boldsymbol{q}$=$\{(y_1 = 1)\}$. Then clause C_1 is satisfied by $\boldsymbol{q}$ and according to Proposition 1, the D-sequent $(\exists x[F], \boldsymbol{q}) \rightarrow \{C_1\}$ holds. Since $F_{\boldsymbol{q}}$=$\{true, C_2\}$, clause C_2 is blocked at variable x. So according to Proposition 3, the D-sequent $(\exists x[F], \boldsymbol{q}) \rightarrow \{C_2\}$ holds.

According to Definition 9, a D-sequent holds with respect to a *particular* $\exists$CNF formula $\exists X[F]$. Proposition 4 below shows that this D-sequent also holds after adding to F any set of resolvent clauses.

Proposition 4: Let $\exists X[F]$ be an $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let G be a CNF formula such that $F \Rightarrow G$. Then if $(\exists X[F], \boldsymbol{q}) \rightarrow R$ holds, $(\exists X[F \wedge G], \boldsymbol{q}) \rightarrow R$ does too.

B. Join Operation for D-sequents

In this subsection, we introduce the operation of joining D-sequents (Definition 11).

Definition 10: Let $\boldsymbol{q'}$ and $\boldsymbol{q''}$ be assignments in which exactly one variable $v \in Vars(\boldsymbol{q'}) \cap Vars(\boldsymbol{q''})$ is assigned different values. The assignment $\boldsymbol{q}$ consisting of all the assignments of $\boldsymbol{q'}$ and $\boldsymbol{q''}$ but those to v is called the **resolvent** of $\boldsymbol{q'}$,$\boldsymbol{q''}$ on v. Assignments $\boldsymbol{q'}$,$\boldsymbol{q''}$ are called **resolvable** on v.

Proposition 5: Let $\exists X[F]$ be an $\exists$CNF formula. Let D-sequents $(\exists X[F], \boldsymbol{q'}) \rightarrow R$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow R$ hold. Let $\boldsymbol{q'}$, $\boldsymbol{q''}$ be resolvable on $v \in Vars(F)$ and $\boldsymbol{q}$ be the resolvent of $\boldsymbol{q'}$ and $\boldsymbol{q''}$. Then, D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ holds too.

Definition 11: We will say that the D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ of Proposition 5 is produced by **joining D-sequents** $(\exists X[F], \boldsymbol{q'}) \rightarrow R$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow R$ at v.

C. Composable D-sequents

In general, the fact that D-sequents $(\exists X[F], \boldsymbol{q}) \rightarrow R'$ and $(\exists X[F], \boldsymbol{q}) \rightarrow R''$ hold does not imply that $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ holds. The reason is that redundancy of R' may be true only when clauses of R'' are in F and vice versa. So, derivation of $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ requires recursive reasoning. Proposition 6 below shows how to avoid recursive reasoning.

Let $\boldsymbol{q}$ and $\boldsymbol{s}$ be assignments to a set of variables Z. Since $\boldsymbol{q}$ and $\boldsymbol{s}$ are sets of value assignments to individual variables of Z, one can apply set operations to them. For instance, $\boldsymbol{s} \subseteq \boldsymbol{q}$ means that $\boldsymbol{q}$ contains the value assignments of $\boldsymbol{s}$. Assignment $\boldsymbol{q} \cup \boldsymbol{s}$ consists of the value assignments that are in $\boldsymbol{q}$ or $\boldsymbol{s}$.

Proposition 6: Let $\boldsymbol{s}$ and $\boldsymbol{q}$ be assignments to variables of F where $\boldsymbol{s} \subseteq \boldsymbol{q}$. Let D-sequents $(\exists X[F], \boldsymbol{s}) \rightarrow R'$ and $(\exists X[F \setminus R'], \boldsymbol{q}) \rightarrow R''$ hold. Then D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ holds.

Definition 12: Let $\boldsymbol{q'}$ and $\boldsymbol{q''}$ be assignments to a set of variables Z. We will say that $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are **compatible** if every variable of $Vars(\boldsymbol{q'}) \cap Vars(\boldsymbol{q''})$ is assigned the same value in $\boldsymbol{q'}$ and $\boldsymbol{q''}$.

Definition 13: Let D-sequent S' be equal to $(\exists X[F], \boldsymbol{q'}) \rightarrow R'$ and S'' be equal to $(\exists X[F], \boldsymbol{q''}) \rightarrow R''$ where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are compatible assignments to $Vars(F)$. We will call S' and S'' **composable** if D-sequent S equal to $(\exists X[F], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R' \cup R''$ holds. From Proposition 6 it follows that S', S'' are composable if D-sequent $(\exists X[F \setminus R'], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R''$ or $(\exists X[F \setminus R''], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R'$ hold.

Although the QE algorithm of [9] derives composable D-sequents we did not explicitly use the notion of composability of D-sequents there. In this paper, we make this important notion more conspicuous.

```
// q is an assignment to Vars(F)
// Ω denotes a set of active D-sequents
// Φ denotes ∃X[F]
// If DCDS returns clause nil (respectively a non-nil clause),
// F_q is satisfiable (respectively unsatisfiable)

DCDS(Φ,q,Ω){
1   if (∃ clause C ∈ F falsif. by q) {
2     Ω := atomic_Dseqs1(Ω, q, C);
3     return(Φ, Ω, C);}
4   Ω := atomic_Dseqs2(Φ, q, Ω);
5   if (all_X_clauses_redund(Φ, Ω)) return(Φ, Ω, nil);
    - - - - - - - - - - - -
6   v := pick_variable(F, q, Ω);
7   (Φ, Ω, C_0) :=DCDS(Φ,q ∪ (v = 0),Ω);
8   if (C_0 ≠ nil) Ω := add_atomic_Dseqs(Ω, q, C_0);
9   Ω_asym := Dseqs_to_be_inactive(F, Ω, v);
10  if (Ω_asym = ∅) return(Φ, Ω, C_0);
11  Ω := Ω \ Ω_asym;
12  (Φ, Ω, C_1) :=DCDS(Φ,q ∪ (v = 1),Ω);
    - - - - - - - - - - - -
13  if ((C_0 ≠ nil) and (C_1 ≠ nil)){
14    C := resolve_clauses(C_0, C_1, v);
15    F := F ∧ C;
16    Ω := atomic_Dseqs1(Ω, q, C);
17    return(Φ, Ω, C);}
18  Ω := merge(Φ, q, v, Ω_asym, Ω, C_0, C_1);
19  return(Φ, Ω, nil);}
```

Fig. 1. $DCDS$ procedure

V. Algorithm Description

In this section, we describe a QE algorithm called ***DCDS*** (Derivation of Clause D-Sequents). $DCDS$ derives D-sequents $(\exists X[F], \boldsymbol{q}) \rightarrow \{C\}$ stating the redundancy of X-clause $C_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$. From now on, we will use a short notation of D-sequents writing $\boldsymbol{s} \rightarrow \{C\}$ instead of $(\exists X[F], \boldsymbol{s}) \rightarrow \{C\}$. We will assume that the parameter $\exists X[F]$ missing in $\boldsymbol{s} \rightarrow \{C\}$ is the *current* $\exists$CNF formula (with all resolvents added to F).

One can omit $\exists X[F]$ from D-sequents because from Proposition 4 it follows that $(\exists X[F], \boldsymbol{s}) \rightarrow \{C\}$ holds no matter how many resolvent clauses are added to F. We will call D-sequent $\boldsymbol{s} \rightarrow \{C\}$ **active** in subspace $\boldsymbol{q}$ if $\boldsymbol{s} \subseteq \boldsymbol{q}$. If $\boldsymbol{s} \rightarrow \{C\}$ is active in subspace $\boldsymbol{q}$, clause $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

A description of $DCDS$ is given in Figure 1. $DCDS$ accepts an $\exists$CNF formula $\exists X[F]$ (denoted as Φ), an assignment $\boldsymbol{q}$ to $Vars(F)$ and a set Ω of D-sequents active in subspace $\boldsymbol{q}$ stating redundancy of *some* X-clauses in $\exists X[F_{\boldsymbol{q}}]$. To simplify description of $DCDS$, by ***X*-clauses of $F_{\boldsymbol{q}}$** we also mean the X-clauses of F satisfied by $\boldsymbol{q}$. On the contrary, an X-clause of F falsified by $\boldsymbol{q}$ is not considered as an X-clause of $F_{\boldsymbol{q}}$. $DCDS$ returns a formula $\exists X[F]$ modified by resolvent clauses added to F (if any), a set Ω of D-sequents active in subspace $\boldsymbol{q}$ that state redundancy of *all* X-clauses in $\exists X[F_{\boldsymbol{q}}]$

and a clause C. If $F_{\boldsymbol{q}}$ is unsatisfiable then C is a clause of F falsified by $\boldsymbol{q}$. Otherwise, C is equal to *nil* meaning that no clause implied by F is falsified by $\boldsymbol{q}$.

The active D-sequents derived by $DCDS$ are composable. That is if $\boldsymbol{s_1} \to \{C_1\}, \ldots, \boldsymbol{s_k} \to \{C_k\}$ are the active D-sequents of subspace $\boldsymbol{q}$, then the D-sequent $\boldsymbol{s}^* \to \{C_1, \ldots, C_k\}$ holds where $\boldsymbol{s}^* = \boldsymbol{s_1} \cup \ldots \cup \boldsymbol{s_k}$ and $\boldsymbol{s}^* \subseteq \boldsymbol{q}$. $DCDS$ achieves composability of D-sequents as follows. As soon as an X-clause $C_{\boldsymbol{q}}$ is proved redundant, it is marked and ignored by $DCDS$, which is equivalent to removing $C_{\boldsymbol{q}}$ from $F_{\boldsymbol{q}}$. So $DCDS$ guarantees that for every path of the search tree leading to a leaf, X-clauses are proved redundant in a particular order. (This order may be different for different paths.) This allows to avoid recursive reasoning where a clause $C'_{\boldsymbol{q}}$ is used to prove redundancy of clause $C''_{\boldsymbol{q}}$ and vice versa. In turn, avoiding recursive reasoning guarantees composability of D-sequents.

A solution to the QE problem in subspace $\boldsymbol{q}$ is obtained by discarding all X-clauses from the CNF formula $F_{\boldsymbol{q}}$ of $\exists X[F_{\boldsymbol{q}}]$ returned by $DCDS$. To build a quantifier-free CNF formula equivalent to Φ, one needs to call $DCDS$ with $\boldsymbol{q} = \emptyset$, $\Omega = \emptyset$.

A. The Big Picture

$DCDS$ consists of three parts separated in Figure 1 by dotted lines. In the first part (lines 1-5), $DCDS$ builds atomic D-sequents i.e. D-sequents for X-clauses whose redundancy can be trivially proved. If all X-clauses are proved redundant in $\exists X[F_{\boldsymbol{q}}]$, $DCDS$ terminates.

If some X-clauses are not proved redundant yet, $DCDS$ enters the second part of the code (lines 6-12). First, $DCDS$ picks a branching variable v (line 6). Then it extends $\boldsymbol{q}$ by assignment to variable v and recursively calls itself (line 7) starting the left branch of v. For the sake of clarity, we assume that $DCDS$ first explores assignment $v = 0$. Once the left branch is finished, $DCDS$ extends $\boldsymbol{q}$ by $(v = 1)$ and explores the right branch (line 12).

In the third part, $DCDS$ merges the left and right branches (lines 13-19). The result of this merging is proving every X-clause redundant in $\exists X[F_{\boldsymbol{q}}]$. For every X-clause $C_{\boldsymbol{q}}$ proved redundant in $\exists X[F_{\boldsymbol{q}}]$, the set Ω contains precisely one active D-sequent $\boldsymbol{s} \to \{C\}$ where $\boldsymbol{s} \subseteq \boldsymbol{q}$. As soon as $C_{\boldsymbol{q}}$ is proved redundant, it is marked and ignored until $DCDS$ enters a subspace $\boldsymbol{q'}$ where $\boldsymbol{s} \not\subseteq \boldsymbol{q'}$ i.e. a subspace where D-sequent $\boldsymbol{s} \to \{C\}$ becomes inactive. Clause $C_{\boldsymbol{q'}}$ is unmarked in $F_{\boldsymbol{q'}}$ signaling that $DCDS$ needs to derive a new D-sequent $\boldsymbol{s'} \to \{C\}$ where $\boldsymbol{s'} \subseteq \boldsymbol{q'}$ stating the redundancy of $C_{\boldsymbol{q'}}$.

B. Building Atomic D-sequents

Procedures *atomic_Dseqs1* and *atomic_Dseqs2* are called by $DCDS$ to compute D-sequents for trivial cases of clause redundancy listed in Section III. We refer to such D-sequents as **atomic**. Procedure *atomic_Dseqs1* is called when formula $F_{\boldsymbol{q}}$ contains an empty clause $C_{\boldsymbol{q}}$ which means that clause C of F is falsified by $\boldsymbol{q}$. For every X-clause $C'_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$ that has no active D-sequent yet, *atomic_Dseq1* generates a D-sequent $\boldsymbol{s} \to \{C'\}$. Here $\boldsymbol{s}$ is the shortest assignment falsifying C.

$atomic\_Dseqs2(\Phi, \boldsymbol{q}, \Omega)\{$
1 $\Omega := \Omega \cup Dseqs(new\_satisf\_clauses(\Phi, \boldsymbol{q}, \Omega))$;
2 $\Omega := \Omega \cup Dseqs(new\_blocked\_clauses(\Phi, \boldsymbol{q}, \Omega))$;
3 return(Ω);

Fig. 2. *atomic_Dseqs2* procedure

If $F_{\boldsymbol{q}}$ does not have an empty clause, procedure *atomic_Dseqs2* shown in Figure 2 is called. It builds D-sequents for X-clauses that became satisfied or blocked in $F_{\boldsymbol{q}}$. Let C be a clause satisfied by $\boldsymbol{q}$. Then D-sequent $\boldsymbol{s} \to \{C\}$ is generated where $\boldsymbol{s} = (v = b)$, $b \in \{0, 1\}$ is the assignment to a variable v satisfying C.

Let $C_{\boldsymbol{q}}$ be blocked in $F_{\boldsymbol{q}}$ at variable $v \in X$ that is not assigned yet. A D-sequent $\boldsymbol{s} \to \{C\}$ stating redundancy of C is built as follows. The fact that $C_{\boldsymbol{q}}$ is blocked at v means that every clause C' of F resolvable with C on v is either satisfied by $\boldsymbol{q}$ or $C'_{\boldsymbol{q}}$ is proved redundant in $F_{\boldsymbol{q}}$. The assignment $\boldsymbol{s}$ is a subset of assignments of $\boldsymbol{q}$ guaranteeing that C' remains satisfied by $\boldsymbol{s}$ or $C'_{\boldsymbol{s}}$ remains redundant in $\exists X[F_{\boldsymbol{s}}]$ and so $C_{\boldsymbol{s}}$ is blocked at v in $F_{\boldsymbol{s}}$. If C' is satisfied by $\boldsymbol{q}$, then $\boldsymbol{s}$ contains an assignment $(v = b)$, $b \in \{0, 1\}$ of $\boldsymbol{q}$ satisfying C'. If C' is not satisfied but $C'_{\boldsymbol{q}}$ is proved redundant in $\exists X[F_{\boldsymbol{q}}]$, then $\boldsymbol{s}$ contains *all* assignments of $\boldsymbol{s'}$ where $\boldsymbol{s'} \subseteq \boldsymbol{q}$ and $\boldsymbol{s'} \to \{C'\}$ is the D-sequent of Ω stating redundancy of $C'_{\boldsymbol{q}}$.

C. Selection of a Branching Variable

Let $\boldsymbol{q}$ be the assignment $DCDS$ is called with. We will say that a **variable** x of X **is redundant** in $\exists X[F_{\boldsymbol{q}}]$ if x is not assigned in $\boldsymbol{q}$ and every $\{x\}$-clause is proved redundant in $\exists X[F_{\boldsymbol{q}}]$. Denote by X_{red} the variables proved redundant in $\exists X[F_{\boldsymbol{q}}]$. Let $Y = Vars(F) \setminus X$. $DCDS$ branches on free (*i.e.*, unassigned) variables of X and Y. Importantly, a free variable $x \in X \setminus Vars(\boldsymbol{q})$ is picked for branching *only* if $x \notin X_{red}$ i.e. $DCDS$ does not branch on variables proved redundant.

Although Boolean Constraint Propagation (BCP) is not shown explicitly in Figure 1, it is included into the *pick_variable* procedure as follows: a) preference is given to branching on variables of unit clauses of $F_{\boldsymbol{q}}$ (if any); b) if v is a variable of a unit clause $C_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$ and v is picked for branching, then the value falsifying $C_{\boldsymbol{q}}$ is assigned first to cause immediate termination of this branch.

To simplify merging results of the left and right branches, $DCDS$ first assigns values to variables of Y (more details are given in Subsection V-E). This means that *pick_variable* never selects a variable $x \in X$ for branching, if there is a free variable of Y. In particular, BCP does not assign values to variables of X if a variable of Y is still unassigned.

D. Switching from Left to Right Branch

$DCDS$ prunes big chunks of the search space by not branching on redundant variables of X. One more powerful pruning technique of $DCDS$ discussed in this subsection is reducing the size of right branches.

Let $\boldsymbol{s} \to \{C\}$ be a D-sequent of the set Ω computed by $DCDS$ in the left branch $v = 0$ (line 7 of Figure 1). We will call this D-sequent **symmetric in v**, if v is not assigned in $\boldsymbol{s}$. Otherwise, this D-sequent is called **asymmetric in v**. Notice that if $\boldsymbol{s}$ is symmetric in v, the D-sequent $\boldsymbol{s} \to \{C\}$ is active in the right branch and so $C_{\boldsymbol{q_1}}$ is redundant in $\exists X[F_{\boldsymbol{q_1}}]$ where

$merge(\Phi, \boldsymbol{q}, v, \Omega_{asym}, \Omega, C_0, C_1)\{$
1 $\Omega := join\_Dseqs\_of\_old\_clauses(v, \Omega_{asym}, \Omega)$;
2 $\Omega := update\_Dseqs\_of\_new\_clauses(v, \Omega)$;
3 if $(v \notin X)$ return(Ω);
4 if $(C_0 \neq nil)$ $\Omega := \Omega \cup \{Dseq(C_0)\}$;
5 if $(C_1 \neq nil)$ $\Omega := \Omega \cup \{Dseq(C_1)\}$;
6 return(Ω);}

Fig. 3. *merge* procedure

$\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$. Denote by Ω_{asym} the subset of active D-sequents that are asymmetric in v. It is computed in line 9. Before exploring the right branch (line 12), the X-clauses of F whose redundancy is stated by D-sequents of Ω_{asym} become non-redundant again. So the set of X-clauses to be considered in the right branch reduces to *only* those with D-sequents from Ω_{asym}. This allows to prune big parts of the search space. In particular, if Ω_{asym} is empty there is no need to explore the right branch. In this case, $DCDS$ just returns the results of the left branch (line 10). Pruning the right branch when Ω_{asym} is empty is similar to non-chronological backtracking well known in SAT-solving [16].

E. Branch Merging

Let $\boldsymbol{q_0} = \boldsymbol{q} \cup \{(v = 0)\}$ and $\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$. The goal of branch merging is to use solutions of the QE problem in subspaces $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$ to produce a solution to the QE problem in subspace $\boldsymbol{q}$. If both $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ are unsatisfiable, this is done as described in lines 14-17 of Figure 1. In this case, the empty clauses $(C_0)_{\boldsymbol{q_0}}$ and $(C_1)_{\boldsymbol{q_1}}$ where C_0, C_1 are clauses returned in the left and right branches respectively are solutions to the QE in subspaces $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$. The empty clause $C_{\boldsymbol{q}}$ where C is the resolvent of C_0 and C_1 added to F (line 15) is a solution to the QE problem in subspace $\boldsymbol{q}$. If, say, $v \notin Vars(C_1)$ and so C_1 cannot be resolved on v, *resolve_clauses* (line 14) returns C_1 itself since C_1 is falsified by $\boldsymbol{q}$. In this case, no new clause is added to F. After C is added, *atomic_Dseqs1* completes Ω by generation of atomic D-sequents built due to presence of a clause falsified by $\boldsymbol{q}$.

Suppose that at least one of formulas $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ is satisfiable. In this case, to finish solving the QE problem in subspace $\boldsymbol{q}$, one needs to make sure that every X-clause is proved redundant in $F_{\boldsymbol{q}}$. This means that every X-clause should have a D-sequent active in subspace $\boldsymbol{q}$ and hence symmetric in the branching variable v. This work is done by procedure *merge* shown in Figure 3 that consists of three steps. In the first step, *merge* takes care of D-sequents of "old" X-clauses that is the clauses that were present in F at the time the value of v was flipped from 0 to 1. For every such X-clause, a D-sequent was derived in the left branch $v = 0$. In the second step, *merge* processes new X-clauses that is X-clauses generated in the right branch $v = 1$. No D-sequents were derived for such clauses in the branch $v = 0$. In the third step, if, say, clause C_0 mentioned above is not equal to *nil*, a D-sequent is generated for C_0 if it is an X-clause.

In the first step, *merge* needs to update only D-sequents of X-clauses that became non-redundant in the right branch because their D-sequents got inactive there (such D-sequents form set Ω_{asym}, see Subsection V-D). Let us denote this set of clauses as G. If a D-sequent of an X-clause C from G returned in the *right* branch is asymmetric in v, then *join_Dseqs_of_old_clauses* (line 1) replaces it with a D-sequent symmetric in v as follows. Let S_0 and S_1 be the D-sequents derived in the left and right branches respectively that state the redundancy of $C_{\boldsymbol{q_0}}$ and $C_{\boldsymbol{q_1}}$. Then *join_Dseqs_of_old_clauses* joins S_0 and S_1 at v producing a new D-sequent S. The latter states the redundancy of $C_{\boldsymbol{q}}$ and is symmetric in v. D-sequent S_1 is replaced in Ω with S.

Let S_1 be symmetric in v. If $F_{\boldsymbol{q_0}}$ was unsatisfiable, then S_1 remains untouched. Otherwise, *join_Dseqs_of_old_clauses* does the following. Let S_1 be equal to $\boldsymbol{s} \rightarrow \{C\}$. First, the right branch assignment $v = 1$ is added to $\boldsymbol{s}$, which makes S_1 asymmetric in v. Then S_1 is joined with S_0 at v to produce a new D-sequent S that is symmetric in v. S replaces S_1 in Ω. The reason one cannot simply keep S_1 in Ω untouched is as follows. As we mentioned above, the composability of D-sequents built by $DCDS$ is based on the assumption that for every path of the search tree, X-clauses are proved redundant in a particular order. It can be shown that using D-sequent S_1 in subspace $\boldsymbol{q}$ may violate this assumption and so break the composability of D-sequents.

Let S be a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ derived in the right branch $v = 1$ where C was generated in this branch i.e. C is a new clause. Such D-sequents are processed in the second step of *merge* by procedure *update_dseqs_of_new_clauses* (line 2). If S is symmetric in v, it simply remains in Ω untouched. Otherwise, S is updated by removing the assignment to v from $\boldsymbol{s}$. One can do this because the clause C is implied by F and has never been used in the left branch. So it can be considered as proved redundant in the left branch.

Finally, *merge* performs the third step (lines 3-5). Notice that if v is not in X, then C_0 or C_1 is not an X-clause. This is because $DCDS$ assigns non-quantified variables before those of X (see Subsection V-C). So the last variable assigned in an X-clause is always a variable of X. Let us assume that $v \in X$ and $C_0 \neq nil$. (In the case $C_1 \neq nil$, *merge* works similarly.) Clause $(C_0)_{\boldsymbol{q}}$ is equal to the unit clause v. Notice that $F_{\boldsymbol{q}}$ does not contain a clause with literal $\overline{v}$ because this would mean that both $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ were unsatisfiable. So, $(C_0)_{\boldsymbol{q}}$ is blocked in $F_{\boldsymbol{q}}$ at variable v. Then an atomic D-sequent is built for C_0 as described in Subsection V-B.

F. Correctness of DCDS

Let $DCDS$ be called on formula $\Phi = \exists X[F]$ with $\boldsymbol{q} = \emptyset$ and $\Omega = \emptyset$. Here is an informal explanation of why $DCDS$ produces the correct result. First, new clauses of F are produced by resolution and so are correct in the sense they are implied by F. In particular, if F is unsatisfiable, $DCDS$ returns an empty clause that is a correct solution to the QE problem. Second, the atomic D-sequents built by $DCDS$ are correct. Third, new D-sequents produced by operation *join* are correct. Fourth, the D-sequents of individual clauses are composable. So when $DCDS$ returns to the root node of the search tree, it derives the correct D-sequent $(\exists X[F], \emptyset) \rightarrow F^X$. Hence, by removing X-clauses from F, one obtains a CNF formula that is a correct solution to the QE problem.

Proposition 7: $DCDS$ is sound and complete.

VI. A Run of $DCDS$ on a Simple Formula

Let $\exists X[F]$ be an $\exists$CNF formula where $F = C_1 \wedge C_2$, $C_1 = \overline{y}_1 \vee \overline{x}$, $C_2 = y_2 \vee x$ and $X = \{x\}$. To identify a particular

$DCDS$ call we will use the corresponding assignment $\boldsymbol{q}$. For example, $DCDS_{(y_1=1,y_2=0)}$ means that the assignments $y_1 = 1$ and $y_2 = 0$ were made at recursion depths 0 and 1 respectively. Originally, assignment $\boldsymbol{q}$ is empty, so the initial call is $DCDS_{(\emptyset)}$. Figures 4, 5 show the work of $DCDS$. We use them to explain the algorithm of $DCDS$. For the sake of simplicity, in this example, we say that clause C_i, $i = 1, 2$ is blocked/redundant in $F_{\boldsymbol{q}}$ meaning that it is clause $(C_i)_{\boldsymbol{q}}$ that is blocked/redundant in $F_{\boldsymbol{q}}$.

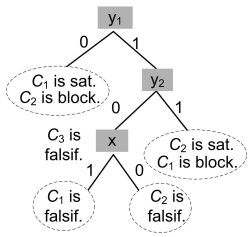


Fig. 4. Search tree built by $DCDS$

Branching variables. Figure 4 shows the search tree built by $DCDS$. Recall that $DCDS$ branches on variables of $Vars(F) \setminus X = \{y_1, y_2\}$ before those of X (see Subsection V-C).

Leaves. The search tree of Figure 4 has four leaf nodes shown in dotted ovals. In each leaf node, either the X-clauses C_1, C_2 are proved redundant or one of them is falsified. For example, C_1 is satisfied and C_2 is blocked, and hence C_1, C_2 are redundant, in leaf $(y_1 = 0)$ and clause C_1 is falsified in leaf $(y_1 = 1, y_2 = 0, x = 1)$.

Generation of new clauses. $DCDS_{(y_1=1,y_2=0)}$ generates a new clause after branching on x. $DCDS_{(y_1=1,y_2=0,x=1)}$ returns C_1 since it is falsified in $F_{(y_1=1,y_2=0,x=1)}$. Similarly, $DCDS_{(y_1=1,y_2=0,x=0)}$ returns C_2 since it is falsified in $F_{(y_1=1,y_2=0,x=0)}$. As described in Subsection V-E, in this case, $DCDS$ resolves clauses C_1 and C_2 on the branching variable x. The resolvent $C_3 = \overline{y}_1 \vee y_2$ is added to F.

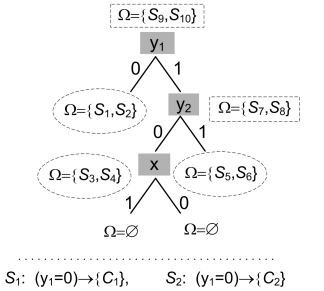


Fig. 5. Derivation of D-sequents

Generation of atomic D-sequents. Figure 5 describes derivation of D-sequents. The atomic D-sequents are shown in dotted ovals. (Dotted boxes show D-sequents obtained by operation *join*.) For instance, C_1 is satisfied by assignment $(y_1 = 0)$ and C_2 is blocked in $F_{(y_1=0)}$. So procedure *atomic_Dseqs2* called by $DCDS_{(y_1=0)}$ generates atomic D-sequents S_1 and S_2 equal to $(y_1 = 0) \rightarrow \{C_1\}$ and $(y_1 = 0) \rightarrow \{C_2\}$ respectively. The atomic D-sequents S_3, S_4 are derived by procedure *atomic_Dseqs1* called by $DCDS_{(y_1=1,y_2=0)}$. As we mentioned above, $DCDS_{(y_1=1,y_2=0)}$ adds clause $C_3 = \overline{y}_1 \vee y_2$ to F. This clause is empty in $F_{(y_1=1,y_2=0)}$ and so D-sequents S_3, S_4 equal to $\boldsymbol{s} \rightarrow \{C_1\}$, $\boldsymbol{s} \rightarrow \{C_2\}$ respectively are generated. Here $\boldsymbol{s} = (y_1 = 1, y_2 = 0)$ is the shortest assignment falsifying C_3.

Switching from left to right branch. Let us consider switching between branches by $DCDS_{(\emptyset)}$ where y_1 is picked for branching. The left branch of $\Omega_{(\emptyset)}$ returns D-sequents S_1, S_2 equal to $(y_1 = 0) \rightarrow \{C_1\}$ and $(y_1 = 0) \rightarrow \{C_2\}$ respectively.

Before starting the right branch $y_1 = 1$, $DCDS_{(\emptyset)}$ computes the set $\Omega^{asym}_{(\emptyset)}$ of D-sequents asymmetric in y_1. Since S_1 and S_2 contain an assignment to y_1, $\Omega^{asym}_{(\emptyset)}$=$\Omega_{(\emptyset)}$ and $DCDS_{(\emptyset)}$ removes S_1, S_2 from Ω. So, before $DCDS_{(y_1=1)}$ is called, both C_1 and C_2 become non-redundant again.

Branch merging. Consider how branch merging is performed by $DCDS_{(y_1=1)}$. In the left branch y_2=0, D-sequents S_3, S_4 are derived that are asymmetric in y_2. In the right branch y_2=1, D-sequents S_5, S_6, also asymmetric in y_2, are produced. By joining S_3 and S_6 at y_2, D-sequent S_7 equal to $(y_1 = 1) \rightarrow \{C_1\}$ is derived. By joining S_4 and S_5 at y_2, D-sequent S_8 equal to $(y_1$=1$) \rightarrow \{C_2\}$ is derived. D-sequents S_7, S_8 state redundancy of C_1, C_2 in $\exists X[F_{(y_1=1)}]$.

Termination. When $DCDS_{(\emptyset)}$ terminates, $F = C_1 \wedge C_2 \wedge C_3$ where $C_3 = \overline{y}_1 \vee y_2$ and composable D-sequents $\emptyset \rightarrow \{C_1\}$ and $\emptyset \rightarrow \{C_2\}$ are derived. By dropping the X-clauses C_1, C_2 one obtains $C_3 \equiv \exists X[C_1 \wedge C_2]$.

Concluding remarks. Due to small size of F, some features of $DCDS$ are not exposed. For instance, the clause C_3 generated by $DCDS$ is not an X-clause. In general, $DCDS$ may produce new X-clauses whose redundancy one needs to prove along with the original X-clauses. Another consequence of using a small example is that D-sequents derived in every node has the form $\boldsymbol{s'} \rightarrow \{C_1\}$ and $\boldsymbol{s''} \rightarrow \{C_2\}$ where $\boldsymbol{s'} = \boldsymbol{s''}$. For larger formulas, assignments $\boldsymbol{s}$ of active D-sequents $\boldsymbol{s} \rightarrow \{C\}$ may be vastly different for different clauses C.

VII. Experimental Results

The objective of experiments was to compare the performance of DDS and $DCDS$ on realistic examples and to give some comparison of D-sequent and BDD based algorithms. (A comparison of DDS with SAT-based QE algorithms is given in [9].) Importantly, in the current implementations of DDS and $DCDS$, D-sequents *are not re-used.* A D-sequent is discarded as soon as it is employed in a join operation.

We believe that reusing D-sequents will drastically boost the performance of both DDS and $DCDS$ like conflict clause re-using speeds up SAT-algorithms. Re-using learned clauses in SAT-solving is beneficial because their involvement in BCP leads to new forced assignments. Re-using D-sequents gives the power of making "asymmetric" decision assignments. If, say, assignment $v = 0$ makes a lot of learned D-sequents active, then this branch may be much easier to finish than branch $v = 1$. Note that a forced assignment can be viewed as *a special case* of an asymmetric decision assignment where one of the two branches terminates immediately. Making asymmetric decision assignments leads to smaller search trees and so re-using D-sequents provides new exciting possibilities. However, tapping this power needs extra research. For that reason, we report experimental results for the algorithms without D-sequent re-using.

TABLE I. *Results on examples solved by MC-DDS or MC-DCDS . The time limit is 2,000s*

model checker	*MC-BDD*	*MC-DDS*	*MC-DCDS*
#solved	193	247	**258**
#timeouts	66	12	1
time for solved by all three (s.)	9,080	11,293	**1,698**

We applied DDS and $DCDS$ to backward model checking. Our implementation was straightforward: DDS and

$DCDS$ were used to compute backward images until an initial state or a fixed point were reached. We will refer to these model checkers as *MC-DDS* and *MC-DCDS*. In experiments, we also used the BDD-based model checker incorporated into the latest version of a tool called PdTrav [22] (courtesy of Gianpiero Cabodi). We ran PdTrav in the backward model checking mode with ternary simulation turned off (as a non-BDD optimization). The other non-BDD optimizations, e.g. computation of the cone of influence, remained active because there was no way to switch them off. Since DDS and $DCDS$ maintain a single search tree, we also forced PdTrav to represent the transition relation by a monolithic BDD. (A D-sequent based QE algorithm *does not have to* build a single search tree e.g. it can employ restarts. However, using restarts requires storing and re-using D-sequents.) We will refer to PdTrav with the options above as *MC-BDD*.

In the experiments, we ran *MC-DDS* , *MC-DCDS* and *MC-BDD* on 758 benchmarks of the HWMCC-10 competition [23] with the time limit of 2,000 seconds. *MC-BDD* solved 374 benchmarks while *MC-DDS* and *MC-DCDS* solved 247 and 258 benchmarks respectively. This is not surprising taking into account the maturity of current BDD algorithms and their re-using of learned information via subgraph hashing. An important fact however is that *MC-DCDS* and even *MC-DDS* solved many problems that *MC-BDD* could not. So, in a sense, *MC-DDS* with *MC-DCDS* and *MC-BDD* favored different subsets of benchmarks.

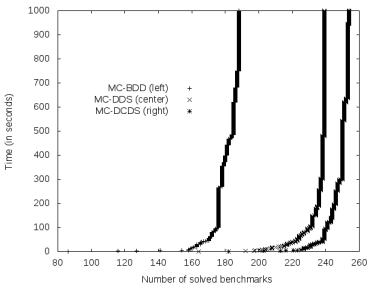


Fig. 6. Performance of model checkers on 259 examples solved by *MC-DDS* or *MC-DCDS*

Table I shows the results of the three model checkers on the 259 benchmarks solved by *MC-DDS* or *MC-DCDS* (i.e. *favored* by the two QE algorithms based on D-sequents). The second line of this table gives the number of benchmarks solved under 2,000s. The third line shows how many examples out of 259 were not solved in the time limit. The last line gives the total time in seconds for the benchmarks solved by all three model checkers. Table I shows that *MC-DCDS* significantly outperformed *MC-DDS*. Besides, a large number of problems solved by *MC-DCDS* were hard for *MC-BDD* . Figure 6 gives the performance of the three model checkers on the benchmarks solved by *MC-DDS* or *MC-DCDS* in terms of the number of problems finished in a given amount of time. *MC-DCDS* (the right line) consistently outperforms *MC-DDS* (the center line). Besides, on this set of benchmarks, both *MC-DDS* and *MC-DCDS* outperform *MC-BDD* (the left line).

Results of all three model checkers on some concrete benchmarks from the 259 benchmark set are given in Table II. Symbol '$*$' marks benchmarks that were not solved in 2,000 s. The column *iterations* show the number of backward images computed by the algorithms before finding a bug or reaching a fixed point.

TABLE II. *Some concrete examples. The time limit is 2,000s.*

benchmark	#latches	#gates	#iterations	bug	MC-BDD (s.)	MC-DDS (s.)	MC-DCDS (s.)
bj08amba4g5	36	13,637	4	no	$*$	113	16
pdtvisbakery3	48	7,514	2	yes	1.3	12	5.1
texasifetch1p5	57	663	21	yes	1.5	368	96
visprodcellp01	78	2,885	5	no	19	7.9	2.3
texasparsesysp1	312	12,173	10	yes	0.7	231	41
bobmiterbm1or	381	3,720	1	yes	0.7	0.1	0.1
pj2003	1175	15,384	3	no	$*$	$*$	252
bobsynthand	3015	15,397	2	no	1.2	0.6	0.6
mentorbm1and	4344	31,684	2	no	$*$	1.8	1.4

Table III sheds light on why $DCDS$ performs better than DDS. It shows results of applying both QE-algorithms to computing all bad states for some benchmarks (the first step of backward model checking). For either algorithm, we give the average size of atomic D-sequents and runtime. By the size of a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ we mean the number of variables assigned in $\boldsymbol{s}$. To make a fair comparison we excluded the atomic D-sequents of length 1 generated when X-clauses got satisfied. Such D-sequents are not built by DDS.

TABLE III. *Relation between average size of atomic D-sequents and runtime*

benchmark	DDS		$DCDS$	
	D-seq size	time (s.)	D-seq size	time (s.)
bc57sensorsp1	19	6.3	13	0.7
boblivea	19	44	10	0.7
bobsmi2c	7.4	96	2.0	9.4
cmugigamax	20	3.6	9.4	3.2
csmacdp2	53	8.1	28	1.5
eijks344	6.0	7.5	2.5	1.6
pdtvissoap2	8.4	82	3.9	3.9
pj2006	7.3	47	1.1	0.4

The results indicate that the size of atomic D-sequents derived by $DCDS$ was smaller. The reason is as follows. When an X-clause is blocked at a variable $v \in X$, $DCDS$ generates a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ where $\boldsymbol{s}$ depends *only* on clauses that can be resolved with C on a variable v. On the contrary, when a variable $v \in X$ is blocked, DDS generates a D-sequent $\boldsymbol{s} \rightarrow \{v\}$ where $\boldsymbol{s}$ depends on *all* clauses that can be resolved on variable v. Such a D-sequent is, in general, much longer than $\boldsymbol{s} \rightarrow \{C\}$.

VIII. Background

The first practical QE algorithms were based on BDDs [3], [4]. Since we focus on SAT-based QE methods we do not discuss these algorithms here. The rest of QE algorithms can be roughly partitioned into two categories. The members of the first category employ various techniques to eliminate quantified variables of the formula one by one in some order [21], [1], [14], [7]. All these solvers face the same problem: there may not exist a good *single* order for variable elimination. This may lead to exponential growth of the size of intermediate formulas.

The solvers of the second category are based on enumeration of satisfying assignments [18], [12], [20], [13]. Since such assignments are, in general, "global" objects, it is hard for such solvers to follow the fine structure of the formula, *e.g.*, such solvers are not compositional [9]. That is they cannot make use of the fact that formula $\exists X_1, X_2[F_1 \wedge F_2]$ where $Vars(F_1) \cap Vars(F_2) = \emptyset$ and $X_i \subseteq Vars(F_i), i = 1, 2$ is equivalent to $\exists X_1[F_1] \wedge \exists X_2[F_2]$.

In [9], we presented a QE algorithm called DDS that employs the machinery of D-sequents based on redundancy

of variables. In a sense, DDS tries to take the best of both worlds. It branches and so can use different variable orders in different branches as the solvers of the second category. At the same time, in every branch, DDS eliminates quantified variables individually as the solvers of the first category, which makes it easier to follow the formula structure. In particular, DDS is compositional (as is $DCDS$).

Identification and removal of redundant clauses is used in preprocessing procedures of QBF-algorithms and SAT-solvers [6], [2]. Redundant clauses are also identified in the inner loop of SAT-solving (inprocessing) [19]. These procedures identify unconditional clause redundancies by recognizing the situations where such redundancies can be easily proved.

Notice that any X-clause C of a CNF formula F can be made redundant in $\exists X[F]$ as follows. Let $v \in Vars(C) \cap X$. Then $\exists X[F] \equiv \exists X[F \setminus \{C\} \cup G]$ where G is the set of all clauses obtained by resolving C with clauses of F on v. In the context of SAT-solving, this fact has been established in [11], [19]. So to make C redundant in $\exists X[F]$, one needs to add all the resolvents of C with clauses of F on a variable of $Vars(C) \cap X$. Hence, one can potentially solve QE by gradually eliminating X-clauses (including the new X-clauses produced by resolution) in some order. Unfortunately, this approach has two fundamental problems. The first problem is similar to that of variable elimination. There may not exist a *single* good order for elimination of X-clauses. The second problem is that global elimination of X-clauses one by one may lead to looping even for small formulas. That is after elimination of a number of clauses one can return to a formula seen earlier e.g. to the original formula. This is because elimination of a clause is accompanied by adding new clauses and so removed clauses may reappear again.

$DCDS$ does not have the problems above. It is not limited by one global order because in different branches of the search tree redundancy of clauses is proved in different orders. $DCDS$ does not have the problem of looping because the branches of a search tree are ordered and the algorithm cannot visit the same state twice.

IX. CONCLUSIONS

We continue to develop a calculus for solving propositional formulas with quantifiers based on the notion of dependency sequents (D-sequents). Previously, we introduced D-sequents recording redundancy of quantified *variables*. In this paper, we present a new type of D-sequents expressing redundancy of *clauses* containing quantified variables. The clause D-sequents are much more powerful that D-sequents based on variable redundancy. We use clause D-sequents to formulate a new algorithm of quantifier elimination called Derivation of Clause Dependency Sequents ($DCDS$).

We experimentally compared $DCDS$ with a QE algorithm employing D-sequents based on variable redundancy in the context of model checking. The experiments showed that $DCDS$ significantly outperformed its counterpart. We also compared a model checker based on $DCDS$ with a BDD-based model checker. We found that there was a noticeable number of benchmarks where $DCDS$ outperformed its BDD-based counterpart. These results are very promising taking into account that the current version of $DCDS$ can be drastically improved e.g. by implementing D-sequent re-using.

ACKNOWLEDGMENT

This work was funded in part by NSF grant CCF-1117184. It was also supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] P. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on sat-solvers. TACAS-00, pages 411–425, 2000.

[2] A.Biere, F.Lonsing, and M.Seidl. Blocked clause elimination for qbf. CADE-11, pages 101–115, 2011.

[3] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] P. Chauhan, E. M. Clarke, S. Jha, J.H. Kukula, H. Veith, and D. Wang. Using combinatorial optimization methods for quantification scheduling. CHARME-01, pages 293–309, 2001.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[7] E.Goldberg and P. Manolios. Sat-solving based on boundary point elimination. HVC-10, pages 93–111, 2010.

[8] E.Goldberg and P.Manolios. Checking satisfiability by dependency sequents. Technical Report arXiv:1207.5014 [cs.LO], 2012.

[9] E.Goldberg and P.Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.

[10] E.Goldberg and P.Manolios. Quantifier elimination by dependency sequents. Technical Report arXiv:1201.5653 [cs.LO], 2012.

[11] A. V. Gelder. Propositional search with k-clause introduction can be polynomially simulated by resolution. In *(Electronic) Proc. 5th Int'l Symposium on Artificial Intelligence and Mathematics*, 1998.

[12] H.Jin and F.Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. DAC-05, pages 750–753, 2005.

[13] J.Brauer, A.King, and J.Kriener. Existential quantification as incremental sat. CAV-11, pages 191–207, 2011.

[14] J.R.Jiang. Quantifier elimination via functional composition. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV-09, pages 383–397, 2009.

[15] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theor. Comput. Sci.*, 223(1-2):1–72, 1999.

[16] J. Marques-Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, 1996.

[17] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[18] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. of CAV-02*, pages 250–264. Springer-Verlag, 2002.

[19] M.Järvisalo, M.Heule, and A.Biere. Inprocessing rules. IJCAR-12, pages 355–370, 2012.

[20] M.K.Ganai, A.Gupta, and P.Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. ICCAD-04, pages 510–517, 2004.

[21] P. Williams, A. Biere, E. Clarke, and A. Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. CAV-00, pages 124–138, 2000.

[22] http://fmgroup.polito.it/index.php/download/.

[23] HWMCC-2010 benchmarks, http://fmv.jku.at/hwmcc10/benchmarks.html.

Interpolation for Synthesis on Unbounded Domains

Viktor Kuncak and Régis Blanc
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

***Abstract*—Synthesis procedures compile relational specifications into functions. In addition to bounded domains, synthesis procedures are applicable to domains such as mathematical integers, where the domain and range of relations and synthesized code is unbounded. Previous work presented synthesis procedures that generate self-contained code and do not require components as inputs. The advantage of this approach is that it requires only specifications as user input. On the other hand, in some cases it can be desirable to require that the synthesized system reuses existing components. This paper describes a technique to automatically synthesize systems from components. It is also applicable to repair scenarios where the desired sub-component of the system should be replaced to satisfy the overall specification. The technique is sound, and it is complete for constraints for which an interpolation procedure exists, which includes e.g. propositional logic, bitvectors, linear integer arithmetic, recursive structures, finite sets, and extensions of the theory of arrays.**

I. Introduction

Software synthesis is an active area of research [5], [13], [14] and has a long tradition [1], [10], [12]. We here pursue synthesis of functions from inputs to outputs that are guaranteed to satisfy a given input/output relation expressed in a decidable logic. Such approach have been referred to as complete functional synthesis [7], [8]. The appeal of this direction is that it synthesizes functions over unbounded domains whenever they exist, and that the produced code is guaranteed to satisfy the specification for the entire unbounded range of inputs. Synthesis procedures for propositional logic, linear rational arithmetic, and Boolean Algebra with Presburger Arithmetic and parametrized coefficients are presented in [6]–[8]. Synthesis procedures for algebraic data types and arrays are presented in [2].

The previous work demonstrated synthesis procedures that generate self-contained code and do not require components as inputs. This approach requires only the input/output specification as the user input. This is in contrast to some of the existing approaches that require components as inputs and enumerate different combinations of the components, checking which ones satisfy a specification. In general, however, synthesis from components is not only a way to simplify the synthesis task, but also a way to control the outcome of synthesis, making the process more predictable. It can be desirable to require synthesis procedures to reuse existing functionality, even if there exists a method to synthesize the system from scratch. For example, using existing components may have expected cost metrics in terms of computational complexity, or market availability. This paper presents techniques that can be used to ensure that a synthesis procedure reuses a given set of components in the synthesized code. The work in reactive LTL synthesis from components [9] deals with stateful reactive components but is limited to finite-state systems and encounters a 2EXPTIME lower bound, whereas we work in the stateless scenario but for infinite domains where we can leverage modern SMT solvers.

Our inspiration comes from generalizing methods such as resynthesis, which have proved useful for generation of combinational circuits [4], [15]. These techniques perform case analysis on boolean variables in the output, which makes them specific to finite domains. We show, however, that such complete technique can be devised for every decidable domain for which interpolation and synthesis procedures exists. This includes bitvector domains, potentially allowing synthesis of circuits at a higher level, as well as the domain of structures used in software, such as recursive algebraic data types, sets, linear integer arithmetic, and arrays. For the approach to work in practice, what is needed are well-behaved interpolation procedures that prefer simpler and computationally shorter interpolants, a requirement that is in any case desirable for interpolation in predicate abstraction refinement [3].

II. Background: Synthesis as Relation Transformation

The starting point for our work is the framework for functional synthesis, as presented most recently in [2], whose notation we follow. For a high-level overview, please consult [7]. A *synthesis problem* is a triple $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$, where $\bar{a}$ is a set of *input* variables, $\bar{x}$ is a set of *output* variables and ϕ is a formula whose free variables are a subset of $\bar{a} \cup \bar{x}$. A synthesis problem denotes a binary relation $\{(\bar{a}, \bar{x}) \mid \phi\}$ between inputs and outputs. The goal of synthesis is to transform such relations until they become executable programs. Programs correspond to formulas of the form $P \wedge (\bar{x} = \bar{T})$ where $\mathrm{vars}(P) \cup \mathrm{vars}(\bar{T}) \subseteq \bar{a}$. We denote programs by $\langle P \mid \bar{T}\rangle$. We call the formula P a *precondition* and call the term $\bar{T}$ a *program term*. We use $\vdash$ to denote the transformation on synthesis problems, so

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!] \tag{1}$$

means that the problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ can be transformed into the problem $[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!]$. The variables on the right-hand side are always the same as on the left-hand side. Our goal is to compute, given $\bar{a}$, one value of $\bar{x}$ that satisfies ϕ. We therefore define the soundness of (1) as a process that refines the binary relation given by ϕ into a smaller relation given by ϕ', without reducing its domain. Expressed in terms of

ISBN 978-0-9835678-3-7/13.

formulas, the conditions become the following:

$$\begin{array}{rcll} \phi' & \models & \phi & \textit{refinement} \\ \exists \bar{x}.\phi & \models & \exists \bar{x}.\phi' & \textit{domain preservation} \end{array}$$

In other words, $\vdash$ denotes domain-preserving refinements of relations. Note that the dual entailment $\exists \bar{x}.\phi' \models \exists \bar{x}.\phi$ also holds, but it follows from *refinement*. Note as well that $\vdash$ is transitive. In most cases we will consider transformations whose result is a program: $[\![\bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle$. The correctness of such transformations reduces to

$$\begin{array}{rcll} P & \models & \phi[\bar{x} \mapsto \bar{T}] & \textit{refinement} \\ \exists \bar{x}.\phi & \models & P & \textit{domain preservation} \end{array}$$

A *synthesis procedure* for a theory $\mathcal{T}$ is given by a set of inference rules and a strategy for applying them such that every formula in the theory is transformed into a program.

III. Interpolation for Synthesis from Components

We next show how synthesis procedure, even for unbounded domains, can leverage interpolation techniques to synthesize a function as a combination of other functions. This enables the user to control the synthesis process by requiring that the desired function is realized as a combination of results of given functions. The technique presented here is inspired by asking whether the finite-state resynthesis techniques from [4], which was experimentally shown to be useful in practice, could be lifted from propositional to the level of first-order theories. The key difficulty is that case analysis on the output, performed in [4], is not possible for infinite-domain theories. We present instead a more general formulation, which works in two stages: 1) construct a quantifier-free input/output constraint describing the implementation of the desired functionality from components, using interpolation for the theory of interest; and 2) synthesize the implementation from the input/output constraint, using the appropriate synthesis procedure.

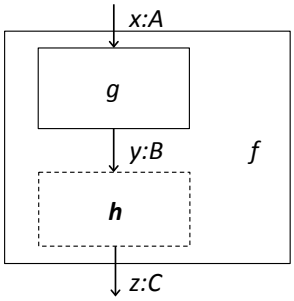
A. Synthesis from Components as a Two-Step Process

Figure 1 summarizes the rules for synthesis from components. The general setup is given by the rule 'COMP' in Figure 1, which is a simple fact of first-order logic with equality. Given a function $f : A \to C$, we encode the available components as another function $g : A \to B$. Note that B can be a cross-product of any number of simpler domains, so g can encode any finite number of component functions. The goal is to express f in terms of the result of g. In other words, we seek a function h such that $f(x) = h(g(x))$. 'COMP' rule gives one way to find such function h:

1) construct a relational description I of the desired h; we say I is a *relational connector* for obtaining f from g.
2) find h as a refinement of the relational connector I.

B. Correctness of Synthesis from a Relational Connector

To see why 'COMP' is correct, let the two assumptions in the rule hold, let x be arbitrary, define z by $z = f(x)$ and b by $b = g(x)$. Then $I(g(x), z)$ by the first assumption, so $I(g(x), h(g(x)))$ by the second assumption. Using the first assumption once again (with $h(g(x))$ as an instance of the universally quantified z), we conclude $h(g(x)) = f(x)$, as desired.



$$\frac{\begin{array}{c} f : A \to C \\ g : A \to B \qquad h : B \to C \qquad I \subseteq B \times C \\ \forall x, z.\ I(g(x), z) \leftrightarrow z = f(x) \\ \forall b.\ (\exists z. I(b, z)) \to I(b, h(b)) \end{array}}{\forall x. f(x) = h(g(x))}\ \text{COMP}$$

$$\frac{\begin{array}{c} f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2 \models I(y_2, z_1) \\ I(y_2, z_1) \models (g(x_2) = y_2 \wedge f(x_2) = z_2 \to z_1 = z_2) \end{array}}{\boxed{\forall x, z.\ I(g(x), z) \leftrightarrow z = f(x)}}\ \text{INT-UNIQ}$$

$$\frac{\begin{array}{c} \text{vars}(I) \subseteq \{b, z\} \\ \boxed{\forall x, z.\ I[b := g(x)] \leftrightarrow z = f(x)} \qquad [\![b\ \langle I \rangle\ z]\!] \vdash \langle P \mid H \rangle \end{array}}{[\![x\ \langle z = f(x) \rangle\ z]\!] \vdash \langle \top \mid H[b := g(x)] \rangle}\ \text{COMP-S}$$

Fig. 1. Synthesis from Components

C. Finding Relational Connector Using Interpolation

We next turn to the problem of finding the relational connector I. The key insight is that a single call to a theorem prover that can compute interpolants [11] is sufficient to find I with the desired property, $\forall x, z.\ I(g(x), z) \leftrightarrow z = f(x)$. This is captured by the 'INT-UNIQ' rule, which stands for "interpolating uniqueness".

To understand the rule, observe that it contains two entailments (universally quantified implications), which, chained together, can be represented as the following property of f and g:

$$\frac{f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2}{g(x_2) = y_2 \wedge f(x_2) = z_2 \to z_1 = z_2}$$

By rearranging the order of assumptions, we can equivalently write this condition as:

$$\frac{\begin{array}{c} g(x_1) = y_1 \qquad g(x_2) = y_2 \qquad y_1 = y_2 \\ f(x_1) = z_1 \qquad f(x_2) = z_2 \end{array}}{z_1 = z_2} \tag{2}$$

This condition states that if g computes the same result on two arguments x_1, x_2, then so does f. Such condition is necessary for the existence of a function h that would enable us to compute $f(x)$ as $h(g(x))$. Indeed, if $g(x_1) = g(x_2)$ then

$h(g(x_1)) = h(g(x_2))$, so we need to have also $f(x_1) = f(x_2)$. Therefore, whenever we can hope to find a function h, we know that the above implication holds. Moreover, if the logic in which f, g are described has the interpolation property, we know that an interpolant I exists. For a decidable logic with interpolation property, rule 'INT-UNIQ' gives an effective algorithm for computing I from f and g.

D. Why Interpolants Precisely Characterize Relational Connectors

We have seen that an I can be found such that the assumptions of the 'INT-UNIQ' rule hold. This ensures that the assumptions of 'INT-UNIQ' rule can be satisfied in practice. We next show the correctness of 'INT-UNIQ': any I that is found in such interpolation process satisfies the conclusion of the 'INT-UNIQ' rule, so it can be used in the 'COMP' rule. Consider the first assumption of 'INT-UNIQ':

$$f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2 \models I(y_2, z_1)$$

Using one-point rule we eliminate y_1 and y_2, replacing them with $g(x_1)$. The result is

$$f(x_1) = z_1 \models I(g(x_1), z_1) \tag{3}$$

Consider the second rule:

$$I(y_2, z_1) \models (g(x_2) = y_2 \wedge f(x_2) = z_2 \rightarrow z_1 = z_2)$$

Using one-point rule we replace y_2 with $g(x_2)$ and replace z_2 with $f(x_2)$, obtaining

$$I(g(x_2), z_1) \models z_1 = f(x_2) \tag{4}$$

By renaming the variables and conjoining (3) and (4), we obtain the desired equivalence:

$$\forall x, y.\ I(g(x), z) \leftrightarrow z = f(x)$$

E. Informal Summary of the Idea

In summary, to express $f(x)$ as $h(g(x))$, we state a necessary condition (2) for f to depend only on the result of g, writing it in a flat form. We then split conjuncts in such a way to separate two uses if f, g between the two sides of the interpolant. Such split leads to interpolants that precisely specify the relationship between the variables y and z, which is the relationship I that we wish to synthesize.

F. From Relation to Function Using Synthesis Procedures

The relational connector I is a relation, so we wish to find a function that refines it. This is where the idea of synthesis using interpolation connects to the framework of synthesis procedures [2], [6]–[8]. The result is a syntactic variant of the rule 'COMP', which we denote 'COMP-S' in Figure 1. The relational connector I is now represented as a formula I with free variables: b (ranging over the set B, the results of g) and z (the desired result of the computation of f and h). Application that was expressed in 'COMP' as $I(g(x), z)$ therefore becomes the substitution $I[b := g(x)]$. Similarly, the desired function h is expressed as a syntactic term H with the free variable b. The condition $f(x) = h(g(x))$ then becomes a synthesis step that transforms $f(x)$ into the term $H[b := g(x)]$ that has x as the free variable.

The key step is $[\![b\ \langle I \rangle\ z]\!] \vdash \langle P \mid H \rangle$, which takes the relational connector and transforms it into a function given by H. In the process, it generates the most general precondition, P. In terms of the rule 'COMP', the condition P corresponds to the condition $\exists z.I(b, z)$, because of the domain-preservation requirement of the "$\vdash$" operator.

Intuitively, because I is only applied to values $g(x)$, the precondition P contains the range of g, so it becomes trivially satisfied in the overall function $h(g(x))$. This allows us to synthesize a program $H[b := g(x)]$ with a trivial, true, precondition $\top$ in the conclusion of the rule.

IV. Further Generalizations

Note that our results apply to any theory for which we have synthesis procedures. The discovery of a relational connector does not even require a synthesis procedure, only interpolation. In practice, we have demonstrated synthesis for many theories and they typically have interpolation [2], [7], [8].

A. Tuples and Passing Inputs

Recall that in the original problem we synthesize $h(y)$ such that $f(x)$ is $h(g(x))$. Note that adding the notion of n-tuples does not change decidability in most cases, because tuple variables can be replaced by individual variables. Thus, we may assume, when convenient, that x is a vector and that g returns a vector.

It can be useful to make some of the coordinates of x directly available to h. To describe this case, we let $x = (x_0, x_1)$ and let $g(x_0, x_1) = (x_0, g_1(x_1))$. Applying the existing rules in Figure 1 to such g we obtain $f(x_0, x_1) = h(x_0, g_1(x_1))$, as desired.

B. Partial Specifications

It may appear at first that the techniques presented here only work when we are given a complete specification of a problem as a function from inputs to outputs. We next show that the framework also supports enforcing arbitrary partial specifications (properties). Indeed, suppose we have a desired specification relation $r \subseteq A \times B$. We view it as a function $f' : A' \rightarrow C'$ where A' is $A \times B$ and C' is $\{0, 1\}$. We then define g to make appropriate transformations on the elements of A, and, for example, pass the elements of B unchanged. Then synthesis of h finds the combination of the outputs of g that enforces the desired properties f', which is again special case of synthesis in our framework.

C. Output Components and Synthesis in Arbitrary Context

So far we considered a problem where given components (g) pre-process the input, which then feeds into the function h that we need to synthesize. It is natural to consider a dual question (see Figure 2): we are given components k that will post-process the result, and we need to synthesize inputs for such components. This problem turns out to be directly

$$\frac{\begin{array}{c} f : A \to C \qquad h : A \to B \qquad k : B \to C \\ \forall x, y.\ I(x, y) \leftrightarrow k(y) = f(x) \\ \forall a.\ (\exists y. I(a, y)) \to I(a, h(a)) \end{array}}{\forall x. f(x) = k(h(x))}$$

$$\frac{\begin{array}{c} \text{vars}(F) \subseteq \{x\} \qquad \text{vars}(K) \subseteq \{y\} \\ [\![x\ \ \langle K = F \rangle\ \ y]\!] \vdash \langle P \mid H \rangle \end{array}}{[\![x\ \ \langle z = F \rangle\ \ z]\!] \vdash \langle P \mid K[y := H] \rangle}\ \text{O-COMP-S}$$

Fig. 2. When components apply before output, we need no interpolation

expressible using synthesis procedures framework, without a quantified synthesis condition. Figure 2 summarizes this case using the semantic rule and the corresponding syntactic synthesis procedure counterpart. In general, the components directly feed into the synthesis procedure invocation. Having pre- and post- processing components simultaneously is therefore solved using the same technique as in the case of pre-processing components alone.

D. Synthesis in Arbitrary Context and Repair

We have concluded that we can do synthesis of missing components that are fed arbitrary inputs, and whose outputs are processed in an arbitrary way. We can therefore solve for h constraints of the form $\forall x.k(h(g(x))) = f(x)$, where f can either check the property or compute value of any other desired type. Such generality enables us to use our framework to repair a given function in two steps: identify the error component, replace it with the unknown component h, then solve for h to enforce the desired constraints. This formulation may help generalize techniques used to solve the engineering change order (ECO) problem [15] to unbounded domains.

E. Synthesizing Multiple Components

Both the argument and the result of h can be a tuple. Therefore, we are able to solve synthesis problems of the form

$$\forall x.k(h_1(g'_1(x), ..., g'_n(x)), ..., h_m(g'_1(x), ..., g'_n(x))) = f(x)$$

This means that we can solve for any number of unknown components. However, note that the results of all components of g are fed into each unknown component. It may be desirable to restrict the inputs of h_i to only a subset of the variables x,

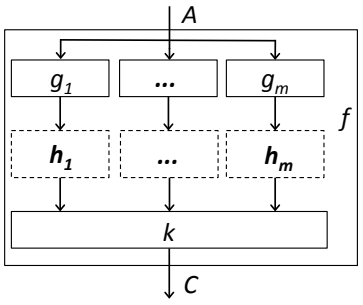


Fig. 3. Our method also handles the more general case

solving instead the problem of the form, (for some different component functions g_j):

$$\forall x.k(h_1(g_1(x)), ..., h_m(g_m(x))) = f(x)$$

To solve such problem we interpolate the following entailment, which, as before, expresses that the result of f only depends on the intermediate results returned by all of g_j:

$$\frac{f(x_1) = z_1 \wedge \bigwedge_{j=1}^{m} g_j(x_1) = y_1^j \ \wedge\ \bigwedge_{j=1}^{m} y_1^j = y_2^j}{f(x_2) = z_2 \wedge \bigwedge_{j=1}^{m} g_j(x_2) = y_2^j \ \rightarrow\ z_1 = z_2}$$

The resulting interpolant is of the form $I(y_2^1, ..., y_2^m, z_1)$; we can easily show that it satisfies, for all x and z,

$$I(g_1(x), ..., g_m(x), z) \leftrightarrow z = f(x)$$

using an entirely analogous proof as in Section III. From such component described using a relation I we can, as before, obtain a function using a synthesis procedure. We thus obtain soundness and completeness for such synthesis of multiple components that are fed distinct parts of the input (Figure 3). In addition to the previous advantages, this generalization enables the user to encode the intuition about independence between variables into the synthesis problem.

Acknowledgements.

We thank Alan Mishchenko for pointing us to existing related work, as well as for his encouraging discussions. We also thank Philippe Suter, Barbara Jobsmtann, Paolo Ienne, and Anna Petkovska for useful discussions.

REFERENCES

[1] Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers (1995)
[2] Jacobs, S., Kuncak, V., Suter, P.: Reductions for synthesis procedures. In: VMCAI (2013)
[3] Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. pp. 459–473 (2006)
[4] Jiang, J.H.R., Lee, C.C., Mishchenko, A., Huang, C.Y.R.: To SAT or not to SAT: Scalable exploration of functional dependency. IEEE Transactions on Computers 59, 457–467 (2010)
[5] Jobstmann, B., Bloem, R.: Optimizations for ltl synthesis. In: FMCAD. pp. 117–124 (2006)
[6] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. Software Tools for Technology Transfer (STTT) (2012)
[7] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. CACM 55(2), 103–111 (2012)
[8] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI. pp. 316–329 (2010)
[9] Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: FOSSACS. pp. 395–409 (2009)
[10] Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. CACM 14(3), 151–165 (1971)
[11] McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
[12] Smith, D.R.: KIDS: A semiautomatic program development system. TSE 16(9), 1024–1043 (1990)
[13] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
[14] Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL. pp. 313–326 (2010)
[15] Wu, B.H., Yang, C.J., Huang, C.Y., Jiang, J.H.R.: A robust functional ECO engine by SAT proof minimization and interpolation techniques. In: ICCAD. pp. 729–734 (2010)

Relational STE and Theorem Proving for Formal Verification of Industrial Circuit Designs

John O'Leary and Roope Kaivola
Intel Corporation
{john.w.oleary, roope.k.kaivola}@intel.com

Tom Melham
University of Oxford
Tom.Melham@cs.ox.ac.uk

***Abstract*—Model checking by symbolic trajectory evaluation, orchestrated in a flexible functional-programming framework, is a well-established technology for correctness verification of industrial-scale circuit designs. Most verifications in this domain require decomposition into subproblems that symbolic trajectory evaluation can handle, and deductive theorem proving has long been proposed as a complement to symbolic trajectory evaluation to enable such compositional reasoning. This paper describes an approach to verification by symbolic simulation, called *Relational STE*, that raises verification properties to the purely logical level suitable for compositional reasoning in a theorem prover. We also introduce a new deductive theorem prover, called *Goaled*, that has been integrated into Intel's Forte verification framework for this purpose. We illustrate the effectiveness of this combination of technologies by describing a general framework, accessible to non-experts, that is widely used for verification and regression validation of integer multipliers at Intel.**

I. Introduction and Motivation

Forte [1] is a formal verification environment, based on symbolic circuit simulation, that is well-established as an effective solution to large-scale, datapath correctness verification at Intel Corporation [2], [3], [4], [5], [6]. Two prominent successes are the verification of the entire execution cluster of the Intel Core 2 Duo [7] and Core i7 processors [8]. Some challenging control-dominated designs have also been verified [9].

The foundation for verification of circuit properties in Forte is *symbolic trajectory evaluation* [10]. Symbolic trajectory evaluation (STE) is a model-checking method powered by symbolic circuit simulation: it computes expressions for circuit outputs in terms of variables that stand for inputs, and checks that the circuit behaviours obtained satisfy temporal logic formulas, computing the exact region of any disagreement. These features give a seamless connection between simulation and verification, as well as comprehensive feedback on failed properties—two key elements of an effective methodology for large-scale formal verification [1], [11].

To control complexity, STE adds a flexible mechanism for partitioned abstraction [12], [13]. But, like any model checker, STE still has limited capacity. Forte therefore complements model checking with a higher-order logic theorem prover of similar design to the HOL system [14]. Theorem proving bridges the gap between big, industrially-important verification tasks and tractable model checking problems. At Intel, Forte is commonly used to provide assurance of functional correctness—rather than 'bug hunting'. (Other tools are used for assertion-based verification.) The verifications tackled are therefore large and highly complex, and almost always require some form of problem decomposition into tractable model-checking cases. A typical high-level correctness statement may decompose in complex ways into tens or even hundreds of individual STE properties. Theorem proving helps assure the verification engineer that these do indeed join up to imply the overall correctness result. Problem decompositions also commonly spin out side conditions that can't be checked by STE itself, but which yield to validation by theorem proving.

The Forte approach is to integrate model checking and theorem proving within the single framework of a functional programming language and its runtime system. A highly engineered implementation of STE is built into the core of the language, with numerous entry points into the internals of the model-checking algorithm provided as user-visible functions. Classically, verification in Forte has been viewed as *programming* activity, in which the functional language is used directly by verification engineers to orchestrate proofs and customize the tools to meet complex verification challenges [1].

More recently, however, the focus at Intel has been shifting to a higher level approach, enabled by two key technical developments: a new theorem prover called *Goaled* and a higher level formulation of property verification by symbolic simulation called *Relational STE*. Goaled is a complete replacement for Forte's original theorem prover, ThmTac [1], [15]. Goaled has a much more complete logical basis than ThmTac and is fully integrated with *reFLect* [16], a principled redesign and implementation of the system's original functional programming language. Relational STE liberates the user from the low-level temporal logic of primitive STE. It allows properties to be expressed in terms of purely logical constraints, which are suitable for compositional reasoning with the Goaled theorem prover.

In this paper, we give the first detailed account of Goaled and Relational STE, and of the higher level approach to verification enabled by these technical developments. We illustrate the approach by describing a general framework for integer multiplier verification that puts the power of Forte into the hands of non-experts, and which is widely used for verification and regression validation of multipliers at Intel.

Intel's deployments of STE and Forte are among the most substantial and sustained formal verification efforts in industry; and, for some time, they were distinctive in general approach. It is therefore encouraging to see the emergence of some

ISBN 978-0-9835678-3-7/13.

impressive results obtained at Centaur Technology [17], [18] using a framework, based around the ACL2 theorem prover, that has many parallels with Forte—as well as some significant differences. We discuss this work in some detail in Section V.

II. The Integrated Goaled Theorem Prover

Theorem proving as a complement to STE model-checking has a long history. The combination was pioneered in the early 1990s in an academic predecessor of Forte called Voss [19]; this was followed by a series of systems that linked STE and theorem proving, culminating in today's mature integration within Forte of the comparatively full-featured theorem prover Goaled—an integrated combination that is seeing increasing use in production verification projects.

HOL-Voss was a mathematically-principled hybrid of symbolic trajectory evaluation and reasoning in the HOL theorem prover [20], [21]. Formal definitions were made in HOL of the mathematical entities of trajectory evaluation and some of the functional programming constructs used in Voss to specify data operations and circuit properties assertions. STE proofs done by Voss could then generate HOL theorems framed in this theory, and one could use HOL to construct higher-level arguments from these. The circuit model remained external to HOL and was represented by an uninterpreted logical constant.

At around the same time, Hazelhurst and Seger developed a simple theorem prover within Voss itself for reasoning in a sound and complete system of inference rules for the temporal logic of STE [22]. The prover was augmented with some ad-hoc 'domain knowledge', such as algebraic rules of multiplication, and had some automatic proof heuristics. The idea was to integrate—in a single framework this time—model checking by symbolic simulation and deductive reasoning about 'deeply embedded' assertions of the STE logic.

A step-change in integration came with the realisation that one might unify the functional *programming* language for scripting STE and the *logical* language within which reasoning is done.[1] Lifted-FL [15] was a deep embedding of the underlying term structure of Voss's functional programming language, called 'FL', within itself. This term structure is essentially the typed λ-calculus, the same as that of the higher-order logic in a theorem prover, such as HOL, that uses Church's formulation of the logic [24]. The syntactic *theorems* of a theorem prover implemented within Voss could now just be quoted fragments of FL—i.e. functional program expressions, of Boolean type, that are manipulated as *data* within the system. Moreover, since logical formulas were now just program expressions, FL's native evaluator could (sometimes) be used to prove them. This gave very fast *proof by evaluation*, as a complement to more laborious deductive inference.

Two generations of a theorem prover called ThmTac [1], [15] were built around this idea and used with STE to verify several challenging industrial circuit designs for which decomposition was essential [1], [25], [26]. This strongly validated the general approach, but the system still had some shortcomings. The logical and programming languages didn't completely align—the most notable gap being pattern matching, which was compiled away in the process of parsing quoted FL expressions. The correct logical treatment of type definitions and recursive function definitions was largely passed over. The core of the theorem prover consisted of a collection of 'trusted tactics', so the logical basis was ad-hoc. More seriously, this meant that theorem proving was biased towards interactive, goal-directed proof [27], limiting its appeal to non-experts. Finally, reasoning was still focussed on circuit properties expressed in the primitive temporal logical of STE.

The next step was an extensive 'rational reconstruction' of the programming language, FL. This was aimed, among other things, at making the connection between the programming and logical language much more principled. It also enabled a host of engineering improvements to the system. The result was *reFLect* [16], the language at the heart of the version of Voss—by now renamed Forte—used today in production formal verification projects at Intel.

ReFLect was designed from the start with theorem proving in mind. The aim was to have precisely the *same* λ-calculus at the core of both the logic and the programming language, and for the theorem prover and the *reFLect* interpreter to use identical internal data structures. This allows flexible (but, for logical soundness, still carefully regulated) intermixture of deduction in the theorem prover and evaluation in the interpreter, including proof by evaluation. To gain confidence in the soundness of this integration, a rather intricate reduction semantics for the language was designed, the rules of which could then be made primitive inference rules of the theorem prover. One innovation was the introduction of function definitions by pattern matching over quoted code, and a major challenge was formulating the right reduction rules for this.

Goaled is a full-featured, but still lightweight, higher order logic theorem prover built in *reFLect* for reasoning about *reFLect* programs. The system is heavily influenced by HOL and HOL Light [28]; it can be seen as a reimplementation of these, but with a radically extended λ-calculus. Following the LCF paradigm, it is built on a trusted core of primitive inference rules, expressed in terms of new formulations of the usual *pre-logic* primitives of α-equivalence, term matching, substitution, and so on. The core includes a full basis for ordinary higher-order logic, full rules for term reduction—including pattern matching over quoted terms—and certain reflection rules for moving between logic and native evaluation.

Function definitions and type definitions in the logic, including quotient types, originate as programming language definitions made at the *reFLect* interpreter level. But, although arbitrary definitions are permitted in the interpreter, definitions are made visible in the logic only after surviving scrutiny. This design choice is motivated by the reality of how *reFLect* is used in practice. *ReFLect* is used for hardware specification and proof scripting, where formal proof is valuable or even essential—but also for tool implementation and general purpose programming, where proof is optional, if it is possible

[1] This idea had long been predated, of course, by the pioneering ACL2 theorem prover [23], which uses Applicative Common Lisp for both the implementation and the logical languages.

at all. Quotient types are defined in *reFLect* by proposing an equality-testing function, and are made visible in the logic upon proof that the equality-testing function is an equivalence relation. Functions that operate on quotient types are made visible in the logic upon proof that they respect equality for that type. Recursive function definitions that pass a syntactic test for primitive recursion are admitted immediately, but in general termination must be proved by exhibiting a well founded relation R and supplying a proof that the arguments to each recursive call decrease with respect to R.

Above this fundamental level, Goaled includes formalized theories of Booleans, the option type, natural numbers, integers, rationals, functions, pairs, and lists. There are also more hardware-oriented theories of fixed-width and variable-length bitvector arithmetic. Proof automation, largely ported from HOL, includes full-featured rewriting and simplification, a meson first order solver, and a Fourier-Motzkin solver for linear arithmetic over $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$. Following common practice in this domain, Goaled has sequent 'tagging' to enable integration with external decision procedures.

All these capabilities were added to Goaled in response to practical reasoning needs. In particular, they support the features of *reFLect* commonly used in verification practice, including overloading, records and quotient types. For the time being, we have found this to be adequate for our domain and a good characterization of 'lightweight' theorem proving in this setting. We needed much more than originally envisaged when, say, ThmTac was introduced. But it is still much less than what mainstream theorem provers such as HOL, Coq [29], or Isabelle/HOL [30] have. This is natural, because the activity we support is more about reasoning about functional programs than, say, doing proofs in algebraic mathematics or reasoning about inductively-defined discrete structures.

III. Relational STE: from Simulation to Logic

Symbolic trajectory evaluation (STE) is a model-checking algorithm that proves properties of circuit behaviour using ternary symbolic circuit simulation. The STE algorithm takes as inputs a circuit and a pair of *trajectory formulas*, called the *antecedent* and *consequent*, that together constitute the property to be checked. Roughly speaking, the intuition is that the antecedent determines certain bits in the initial state and provides stimuli to selected circuit inputs at certain points within a bounded period of time. The consequent specifies the values expected to appear on selected circuit nodes as a response, while the circuit model is simulated.

A successful run of STE establishes a *trajectory assertion* saying that any execution of the circuit that conforms to the antecedent also satisfies the consequent. In essence, STE checks that circuit behaviour has simple stimulus-response properties, framed within a finite window of time. In addition, there is a mechanism for abstraction of circuit behaviour in which circuit nodes can carry 'unknown' values. This is overlaid by a symbolic representation for groups of properties that allows relationships between values on different circuit nodes to be expressed. Together, these provide a means by which families of abstractions, each covering only part of the circuit's behaviour, can be checked simultaneously. This mechanism for partitioned abstraction is called *symbolic indexing* [12] and can sometimes achieve dramatic efficiency gains [13], [31].

In the classical formulation of STE, the antecedent and consequent are written in a very simple linear-time temporal logic. In Forte, these formulas are represented concretely by lists of 5-tuples of the form

$$(guard, node, value, start, end)$$

where $guard$ and $value$ are formulas of propositional logic (usually BDDs, but a non-canonical representation aimed at SAT is supported too), $node$ is a node name (a string), and $start$ and end are non-negative integers. The meaning is that if $guard$ holds, then $node$ has $value$ from simulation cycle $start$ up to but excluding simulation cycle end.

ThmTac and earlier theorem provers for STE provided a specialised deductive system for compositional reasoning about the trajectory assertions in the form just introduced. In essence, lists of 5-tuples constituted a 'deep embedding' of a syntax of trajectory assertions; and rules were added to higher order logic that constituted an axiomatic theory of the embedded STE simulation logic. Consequently, the deductive system for circuit properties was not well integrated with the ordinary higher order logic of these theorem provers.[2]

A. Circuit Execution Semantics and STE Formulas

Let us introduce some basic definitions. A *circuit* is a well-formed interconnection of combinational gates and sequential elements, such as flip-flops and latches. An *execution* is a function of type $(\mathit{string} \times \mathit{num}) \to \mathit{bool}$ that assigns to each circuit node, named by the string, a Boolean value at each point in time, represented by a natural number. The *behaviour* of a circuit is a predicate on executions—or, equivalently, a set of executions.

We shall assume the existence of a function

$$[\![ckt]\!] :: ((\mathit{string} \times \mathit{num}) \to \mathit{bool}) \to \mathit{bool}$$

that gives us the behaviour of a circuit, i.e. a predicate determining whether a given execution e is consistent with the circuit. Analogously, we also write $[\![ant]\!]$ for a predicate specifying whether an execution e is consistent with the five-tuple list ant. We are deliberately vague about how a circuit is represented concretely, but note that if a representation is chosen it is a relatively simple matter to define the function $[\![\_]\!]$ mathematically. In essence, one would use the classical 'relational' approach that is well-known from hardware modelling in higher-order logic [32].

B. Relational STE

STE has proven to be an extremely useful verification engine in practice, and has been the key enabler for most of Intel's formal verification success stories. Nevertheless, the

[2]Some bridges between the two levels were, however, provided by certain quantifier rules and axioms about 'parametric' encoding of assumptions. See Section VII of [1] for details.

language of trajectory assertions severely limits the classes of properties that can be expressed natively. In effect, trajectory assertions require a specification to be functional: given inputs, the specification dictates the values the outputs shall have, potentially under some do-care conditions.

Many informal specifications occurring in practice do not fall into this category, the simplest one being 'nodes a and b are mutually exclusive'. Such relational specifications become especially important as the abstraction level of specifications rises. For example, a natural specification of a scheduler might say 'an operation with its sources ready will be scheduled for execution', while intentionally leaving open the selection between different ready operations. When verifying relational specifications with STE, the established practice has been to use STE as a symbolic simulation engine only, and to write ad hoc FL code to compute the satisfaction of the specification on the basis of simulation values queried from the STE trace.

Relational STE, or rSTE in short, has been crafted as a systematic solution to the problem of expressing and verifying general relational specifications, while retaining the use of STE as the underlying symbolic simulation engine.

In what follows, we give an overview of the technical underpinnings of rSTE. The notation we use is an idealization of the Goaled higher order logic. As discussed in Section II, phrases of this logic are simultaneously *reFLect* programs, so we shall employ a mixture of functional programming and logical notation. The reader is spared the concrete syntax of the actual Forte implementation, which for historical reasons is similar to that of the original 'Edinburgh' ML [33].

The basic building blocks for rSTE specifications are called *constraints*. Conceptually, a constraint is simply a predicate on circuit executions. Technically, a constraint c consists of three parts: name, predicate and signature, denoted by $\texttt{name}\ c$, $\texttt{pred}\ c$ and $\texttt{sig}\ c$, respectively. The name is simply a string that is used to identify the constraint for user convenience, e.g for informational messages. The predicate is a function

$$\texttt{pred}\ c :: ((\mathit{string} \times \mathit{num}) \to \mathit{bool}) \to \mathit{bool}$$

and the signature is a list of $\mathit{string} \times \mathit{num}$ pairs. The predicate refers to a finite collection of individually specified circuit nodes and points of time, conceptually querying their values in a circuit execution given to the predicate as an argument, and computes a Boolean function of these values. The signature lists all the nodes referred to by the predicate, and the times at which their value is accessed.

For example, the constraint shown below could be used to express the informal property that 'circuit nodes a and b are mutually exclusive at time point 2'.

$$\begin{array}{l}\texttt{CONSTR}\ \text{“ab\_mutex”} \\ \qquad (\lambda e.\neg((e(\mathsf{a},2)) \wedge (e(\mathsf{b},2)))) \\ \qquad [(\mathsf{a},2),(\mathsf{b},2)]\end{array}$$

We extend the predicate function pred to a constraint list $cl = [c_1, \ldots, c_n]$, implicitly considered conjuncted, by:

$$\texttt{predl}\ cl\ e \stackrel{\text{def}}{=} (\texttt{pred}\ c_1\ e) \wedge \ldots \wedge (\texttt{pred}\ c_n\ e)$$

The user interface to rSTE is through a *reFLect* function

$$\texttt{rSTE}\ \mathit{ckt}\ \mathit{antc}\ \mathit{antv}\ \mathit{cin}\ \mathit{cout}\ \mathit{opts}$$

The first two arguments to rSTE, the circuit ckt and the constant antecedent $antc$, describe the 'static' aspects of the verification task. Here $antc$ is a five-tuple list, exactly as used in classical STE; but it is used in rSTE only to set constant value assignments, e.g. clock patterns, testability signals, etc., that are shared by all verification tasks on the circuit. In classical STE, the antecedent is also used to assign symbolic variables to circuit nodes; in rSTE this aspect is separated out as a distinct variable binding antecedent $antv$, which allows the user only to bind positive instances of distinct symbolic variables to circuit nodes. Every circuit node is mentioned in $antc$ or $antv$ or is assigned an unknown 'X' value at the start of the simulation, so this covers the full state-space. For more discussion on symbolic variable bindings, see [34].

The main logical content of an rSTE verification task is described with the input and output constraint lists cin and $cout$. The meaning is that if the input constraints cin hold, then circuit behaviour under simulation will satisfy the output constraints $cout$. In theory, the constraints could be combined into an implication $cin \Rightarrow cout$. In practice, however, verification of each element in $cout$ may be carried out separately to alleviate complexity. Separating out the constraints cin can also allow some them to be injected into the symbolic simulation using parametric substitution [26], futher improving efficiency.

Finally the options list $opts$ gives the user fine-grained control over how rSTE carries out the verification task. For example, the computation may require use of a parametric representation of boolean functions [26], a specific circuit abstraction method, or certain simulation or constraint satisfaction engines: e.g. BDD-based analysis vs SAT-based analysis. The user specifies all such choices through the rSTE options.

Internally, the *reFLect* function rSTE uses the constant and variable binding antecedents $antv$ and $antc$ as well the user-given options $opts$ to construct a classical STE antecedent. It then carries out symbolic simulation with STE using this antecedent as stimulus. Following the symbolic simulation, an execution $\hat{e}$ is available that 'reads off' symbolic values on circuit nodes at any specified times. To establish truth or falsehood of the rSTE assertion, the implication

$$(\texttt{predl}\ \mathit{cin}\ \hat{e}) \implies (\texttt{predl}\ \mathit{cout}\ \hat{e})$$

is evaluated over $\hat{e}$, either using BDDs or a SAT solver, depending on user options.

The relational formulation of symbolic trajectory evaluation preserves the power of the underlying STE algorithm while enabling much richer specifications—in particular ones describing relations between nodes values rather than just functions. Since its introduction, rSTE has been the workhorse of datapath verification at Intel; many thousands of individual operations, from the very simple to the very complex, have been verified in several microprocessor families and over the course of several generations.

In the classical theory of STE, the 'fundamental theorem' relates a successful run of the symbolic simulator to the logical property it establishes. For rSTE, the fundamental theorem has a particularly elegant formulation:

$$\begin{array}{l}\forall ckt\ antc\ antv\ cin\ cout\ opts\ .\\ \quad \texttt{rSTE}\ ckt\ antc\ antv\ cin\ cout\ opts \Longrightarrow\\ \quad\quad \forall e.[\![ckt]\!]\ e \wedge [\![antc]\!]\ e \Longrightarrow\\ \quad\quad\quad (\texttt{predl}\ cin\ e) \Longrightarrow\\ \quad\quad\quad (\texttt{predl}\ cout\ e)\end{array}$$

Most important for the purpose of this paper, the relational formulation eliminates the need to use specialized STE inference rules and apparatus for temporal reasoning. The relational formulation makes it 'just' higher order logic. Reasoning about functional and temporal aspects of circuit behavior takes place in a uniform framework: higher order logic. Indeed, rSTE itself is a function that can be reasoned about in higher order logic. In Section IV, we illustrate a practical application of this: reasoning about automatically generated specifications and automatically generated calls to rSTE.

Notice that the symbolic variables bound by *antv* do not appear in the correctness property inferrable from a successful run of rSTE. This is intentional: we use symbolic simulation and computation only as a means to the end goal of verifying universally quantified claims. The identity of the symbolic variables and the precise bindings do not matter, as long as distinct variables are bound to distinct circuit nodes and times, which rSTE checks automatically.

IV. A Framework for Integer Multiplication

Figure 1 shows a Booth multiplier. Its principles of operation are simple. First, one of the operands, S1 say, is Booth encoded: N Booth coefficients $\texttt{BE}_i(\texttt{S1})$ are computed such that

$$-2^k < \texttt{BE}_i(\texttt{S1}) < 2^k.$$

for $0 \leq i < N$ and $k > 0$. A given multiplicand S1 has many valid Booth encodings, but the Booth coefficients are always required to satisfy

$$\texttt{S1} = \sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{S1}) \times 2^{ki}. \tag{1}$$

The quantity 2^k is called the *radix*.

Second, a set of N *partial products* is computed, one for each Booth coefficient:

$$\texttt{PP}_i = \texttt{BE}_i(\texttt{S1}) \times \texttt{S2} \tag{2}$$

for $0 \leq i < N$.

Finally, the N partial products are shifted and summed to yield the product PROD:

$$\texttt{PROD} = \sum_{i=0}^{N-1} \texttt{PP}_i \times 2^{ki} \tag{3}$$

Automatic input-to-output verification of even a moderately-sized multiplier is beyond the capacity of the BDD- and SAT-based verification approaches commonly deployed in industry.

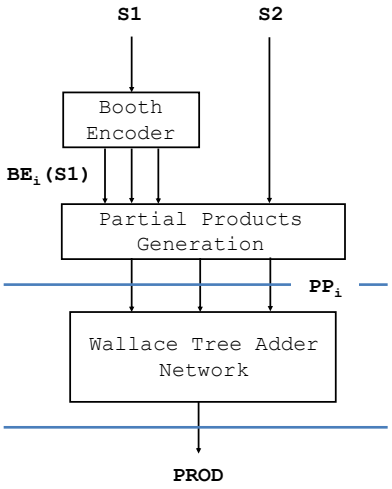


Fig. 1. A Booth Multiplier.

But verification of Equations (1), (2), and (3) is tractable and can be done automatically using rSTE. Once these are established it will require only straightforward algebraic reasoning to prove, on paper or in Goaled, that

$$\texttt{PROD} = \texttt{S1} \times \texttt{S2} \tag{4}$$

Consider now the task of verifying an actual circuit implementation of a Booth multiplier. Typically, a circuit expects to receive a valid clock pattern and the assertion of some interface control signals requesting the execution of a multiplication operation. In rSTE verification, we would fix a constant reference time for the start of the operation, and code the expected clock and control signal patterns by a constant antecedent antc. The circuit will read source data values on designated signals at some fixed delay after the start of the operation. Then, Booth encodings and partial products will be computed, and partial products summed together to produce a final product on designated result signals at some later time.

To map the conceptual proof stages above to an actual circuit implementation, we need to first identify the circuit signals for both sources S1 and S2, the partial products and the final product, with the appropriate timing relative to the fixed start of the operation, and code these as *string* × *num* lists s1, s2, $\texttt{pp}_i$ and prod, respectively. The function s2i interprets the values on such lists, relative to a given circuit execution, as integers using two's complement encoding.

We construct an rSTE constraint to check correctness of the Booth coefficients using Equation (1) as specification:

$$\begin{array}{lcl}\texttt{eqn1}(x) & \stackrel{\text{def}}{=} & x = \sum_{i=0}^{N-1} \texttt{BE}_i(x) \times 2^{ki}\\ \texttt{boothc} & \stackrel{\text{def}}{=} & \texttt{CONSTR}\ \text{“boothOK”}\\ & & \quad (\lambda e.\ \texttt{eqn1}\ (\texttt{s2i}\ e\ \texttt{s1}))\\ & & \quad \texttt{s1}\end{array}$$

As before, we use ordinary mathematical notation in definitions; *reFLect* notation differs in style but not in substance.

We now use rSTE to verify that the constraint `boothc` holds for all executions of the circuit. We will use the variable binding antecedent `antv` to assign distinct symbolic variables to all the source data signals at the time the circuit is expected to read their values, and execute the rSTE call

$$\texttt{rSTE ckt antc antv [ ] [boothc] opts.}$$

Success of this call allows us to conclude, via the fundamental theorem, that

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow (\texttt{predl}\ [\,]\ e) \Longrightarrow (\texttt{predl}\ [\texttt{boothc}]\ e)$$

By expanding the definition of `predl` and employing a few of Goaled's standard theorems about lists, this simplifies to

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \texttt{pred boothc}\ e$$

Expanding the definitions of `boothc` and `pred` plus a little more rewriting yields

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \texttt{eqn1(s2i}\ e\ \texttt{s1)}.$$

Finally, expanding the definition of the auxiliary function `eqn1` yields a theorem asserting that the Booth coefficients bear the correct relationship to the circuit nodes `s1`.

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \implies \texttt{s2i}\ e\ \texttt{s1} = \textstyle\sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times 2^{ki}.$$

In similar fashion, we define constraints corresponding to Equations (2) and (3):

$$\begin{aligned}
\texttt{eqn2}_i(w_i, x, y) &\stackrel{\text{def}}{=} w_i = \texttt{BE}_i(x) \times y \\
\texttt{ppc}_i &\stackrel{\text{def}}{=} \texttt{CONSTR}\ \text{“ppiOK”} \\
&\qquad (\lambda e.\, \texttt{eqn2 (}\ \texttt{s2i}\ e\ \texttt{pp}_i, \\
&\qquad\qquad \texttt{s2i}\ e\ \texttt{s1}, \\
&\qquad\qquad \texttt{s2i}\ e\ \texttt{s2}\ \texttt{))} \\
&\qquad (\texttt{s1 @ s2 @ pp}_i) \\
\texttt{eqn3}(z, w) &\stackrel{\text{def}}{=} z = \sum_{i=0}^{N-1} w_i \times 2^{ki} \\
\texttt{prodc} &\stackrel{\text{def}}{=} \texttt{CONSTR}\ \text{“prodOK”} \\
&\qquad (\lambda e.\, \texttt{eqn3(}\ \texttt{s2i}\ e\ \texttt{prod}, \\
&\qquad\qquad \texttt{map}\ (\texttt{s2i}\ e)\ \texttt{pp}\ \texttt{))} \\
&\qquad (\texttt{prod @ flat pp})
\end{aligned}$$

With these definitions in hand we execute the remaining rSTE runs. There are $N + 2$ in all, one for `boothc`, one instance of $\texttt{ppc}_i$ for each of the N partial products, and one final run to check `prodc`. For all except the final run we use the variable binding antecedent to assign variables to the source data signals; in the final run we use it to assign distinct symbolic variables to the partial product signals.

Using a BDD variable ordering that aligns the bits in partial products according to their position in the summation, we can handle verification of most Wallace tree adders occurring in Intel designs, up to extended precision floating point multipliers. For a minority of designs, a further cut-point in the middle of the Wallace tree is needed to manage BDD complexity, and Equation (3) is split into two obligations.

If all of these verification runs are successful, we can use the fundamental theorem of rSTE and simple rewriting and logical reasoning to conclude that

$$\begin{aligned}
&\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \implies \\
&\quad (\texttt{s2i}\ e\ \texttt{s1} = \textstyle\sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times 2^{ki}) \wedge \\
&\quad (\textstyle\bigwedge_{i=0}^{N-1} \texttt{s2i}\ e\ \texttt{pp}_i = \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times \texttt{s2i}\ e\ \texttt{s2}) \wedge \\
&\quad (\texttt{s2i}\ e\ \texttt{prod} = \textstyle\sum_{i=0}^{N-1} (\texttt{s2i}\ e\ \texttt{pp}_i) \times 2^{ki})
\end{aligned}$$

From here, routine arithmetic reasoning yields a theorem asserting the correctness of the multiplier implementation:

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \implies (\texttt{s2i}\ e\ \texttt{prod}) = (\texttt{s2i}\ e\ \texttt{s1}) \times (\texttt{s2i}\ e\ \texttt{s2})$$

Note that these results were obtained using only standard logical reasoning, without the use of purpose-built inference rules for STE. This is due to the direct representation of rSTE constraints in the logic of Goaled, and our formulation of the fundamental theorem of rSTE that directly exposes the logical import of each successful rSTE run.

A. A General Framework for Multipliers

The method outlined above suffices to verify the correctness of one multiplier. Wide deployment across a large corporation presents additional challenges. Surface details like signal names and their timing vary from design to design, as do radix and operand widths. Designs differ more fundamentally in how they choose to encode Booth coefficients and partial products and how they represent flags and exceptional conditions. Many design-specific quirks are handled by customizing functions like `s2i` that extract values from the circuit trace, allowing us to view the circuit as if it was a vanilla design. If an implementation feature cannot be 'explained away' like this, the reference model is generalized to handle it. In addition, each *design organization* has its own validation and regression practices and software that supports them.

To address this diversity of designs and design environments, we have developed over the last decade a general framework for multiplier verification. The notion of constraints is generalized to allow predicates over arbitrary domains, with the constraints over circuit executions described in Section III being a special case. Abstraction mappings between constraints are used to separate the essence of the specifications and proofs shown above from accidental details of particular designs. The framework is designed to be configured and used by non-experts, who are responsible for supplying design details (signal names, representation of partial products, and so on) and setting configuration parameters (for example, radix and operand width). Behind the scenes, a sophisticated set of *reFLect* scripts arranges for design-specific specifications to be generated and orchestrates the necessary runs of rSTE.

Although the scripts are written with care, they are under continual refinement—each new design and design environment introduces some wrinkle—making it impossible to prove correctness of the scripts once and for all. There is a very real

risk that script errors will result in generation of incorrect specifications or an incomplete set of rSTE runs. To verify correct operation of the scripts, we have integrated Goaled with our multiplier verification framework. During nightly RTL regression, Goaled analyzes the scripts at source level to determine what specifications are generated and what rSTE runs are executed. A proof is programatically constructed, along the lines shown above, that these specs and rSTE runs are sufficient to ensure correctness of the circuit. This capability is currently deployed in a 'live' CPU design project.

Goaled also plays a more traditional role in our multiplier verification framework. Several side conditions arise in our compositional proofs that would be difficult or impossible to prove using BDDs or SAT. For example, this assumption is required in the proof of the Wallace tree adder:

$$\min\{\overline{\mathtt{S1S2}}, \overline{\mathtt{S1}}\underline{\mathtt{S2}}, \underline{\mathtt{S1}}\overline{\mathtt{S2}}, \underline{\mathtt{S1S2}}\} \leq \sum_{i=0}^{N-1} \mathtt{PP}_i \times 2^{ki}$$

where $\overline{x}$ ($\underline{x}$) denotes the largest (smallest) value of the bitvector x. A Goaled proof is immediate from Equations (1) and (2).

V. Related Work

Integration of model-checking and theorem proving was proposed as early as the mid 1990s [35], and many experiments in combining the techologies have been reported. One of these, ACL2SIX [36], integrates the ACL2 theorem prover and IBM's SixthSense verification tool. The combination was used to verify a pipelined 53×43 bit multiplier, by a decomposition strategy similar to the example presented in Section IV. Some more general verification frameworks that effectively combine two technologies have also been designed, two prominent examples being Prosper [37] and SAL [38].

The published research on applied formal verification most closely related to Forte is work on a verification toolflow used at Centaur Technology to help ensure correctness of their X86-compatible microprocessors [17], [18]. Developed and deployed by a team of engineers and scientists at Centaur and UT Austin, the framework integrates several reasoning tools and is based around the well-established ACL2 theorem prover [23]. As with Forte at Intel, a prominent application is the verification of floating-point and integer arithmetic hardware. Of course ACL2 itself, and its predecessor the Boyer-Moore theorem prover, have a long history of successful application to hardware verification [39], [40].

A notable feature of the Centaur framework is that it is built on top of publicly-available software tools: ACL2 itself and special-purpose tools such as the ZZ framework [41] and ABC [42]. The impressive verification results cited in [17], [18] and [43] show that a robust and practical toolflow of considerable capacity can be built in this way. By contrast, Forte is an in-house tool, highly engineered and optimised through years of use on challenging problems at Intel.

The two frameworks have many features in common, at least at a general level: symbolic circuit simulation is a central technology for generating circuit properties; individual properties of a proof are composed together in a theorem prover, to build up more complete verifications; and both systems are embedded in a general purpose programming language, so they can be extended and customised. Both tools can read and give semantics to large circuit models at either gate or transistor levels. Proof regression is a common verification activity carried out with both frameworks.

A significant difference between Forte and the Centaur framework is the depth of integration of circuit simulation. In Forte, the symbolic simulation algorithm is built in to *reFLect* and not exposed to reasoning at the level of the Goaled theorem prover. There are *reFLect* functions that can be used to explore and manipulate the structure of the circuit model, and access its state-transition semantics. But for efficiency the simulator itself is hard-coded into the internals of *reFLect*. Although the STE algorithm has been independently verified [44], there are no plans to verify the highly engineered Forte internal simulator. In the Centaur system based on ACL2, the symbolic circuit simulator is written in ACL2 itself. It is hence available as a formal object about which proofs can be done—and, indeed, has been verified correct [45].

The successful industrial deployment of two major verification frameworks, Forte at Intel and the ACL2-based tools at Centaur Technology, show that this idea has come of age industrially. Moreover the parallels between the two systems, each quite different from the other in numerous matters of detail, strengthens the conclusion that this kind of architecture represents a general solution in this important domain.

VI. Summary and Prospects

This paper has described what we hope to be the basis for a step-change in the exploitation of theorem proving as a complement to symbolic simulation for compositional verification of circuit designs. Relational STE raises the level of the properties obtained from symbolic trajectory evaluation to pure logic. Compositional reasoning can then be done straightforwardly in higher-order logic, rather than with specialised STE inference rules. A 'lightweight' theorem prover, Goaled, has been designed for this purpose and tightly integrated into Forte, the STE programming environment used at Intel. The utility of this approach is exemplified by a general tool for validation of integer multipliers that soundly automates complex proof decompositions for non-expert users, entirely 'hiding' the theorem proving support for this.

Future work on Goaled includes development of a theory of floating point operations at bit level. This is intended for certification of conformance to IEEE Standard 754 of 'reference models' of floating point algorithms expressed as *reFLect* programs. Valuable results of this kind been obtained in the past using ThmTac [1]; it is hoped that a capability for this in Goaled will aid in maintaining the certification of reference algorithms as they become more complex over time.

Future work on Relational STE includes fully integrating SAT-based symbolic trajectory evaluation into the framework, alongside BDDs. This is largely a matter of engineering. More challenging from a research perspective will be the incorporation of *symbolic indexing*, a flexible and somewhat

subtle mechanism for abstraction in STE, into the Relational STE flow. This may leverage past work on abstraction transformations [46] and automatic symbolic indexing [13].

REFERENCES

[1] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. 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, Sept. 2005.

[2] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technical Journal*, First quarter, 1999.

[3] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of Pentium 4 floating-point divider," *Int. J. on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003.

[4] R. Kaivola and N. Narasimhan, "Formal verification of the Pentium-4 multiplier," in *High-Level Design Validation and Test*. IEEE, 2001, pp. 115–122.

[5] A. Slobodová and K. Nagalla, "Formal verification of floating point multiply add on itanium processor," in *Fifth International Workshop on Designing Correct Circuits: Barcelona*. ETAPS 2004, Mar. 2004.

[6] A. Slobodová, "Formal verification of hardware support for advanced encryption standard," in *2008 Formal Methods in Computer Aided Design*. IEEE, 2008, pp. 1–4.

[7] A. Flaisher, A. Gluska, and E. Singerman, "Case study: Integrating FV and DV in the verification of the Intel Core 2 Duo microprocessor," in *Formal Methods in Computer Aided Design*. IEEE, 2007, pp. 192–195.

[8] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, 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*, ser. LNCS. Springer-Verlag, 2009, pp. 414–429.

[9] R. Kaivola, "Formal verification of pentium® 4 components with symbolic simulation and inductive invariants," in *Computer Aided Verification*, ser. LNCS, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer-Verlag, 2005, pp. 170–184.

[10] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.

[11] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical formal verification in microprocessor design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, 2001.

[12] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*. ACM Press, June 1991, pp. 397–402.

[13] S. Adams, M. Björk, T. Melham, and C.-J. Seger, "Automatic abstraction in symbolic trajectory evaluation," in *FMCAD*, 2007, pp. 127–135.

[14] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[15] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Lifted-FL: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 1690. Springer-Verlag, 1999, pp. 323–340.

[16] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 16, no. 2, pp. 157–196, Mar. 2006.

[17] W. A. Hunt, Jr., S. Swords, J. Davis, and A. Slobodova, *Use of Formal Verification at Centaur Technology*. Springer-Verlag, 2010, pp. 65–88.

[18] A. Slobodová, J. Davis, S. Swords, and W. Hunt, "A flexible formal verification framework for industrial scale validation," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE, 2011, pp. 89–97.

[19] C.-J. H. Seger, "Voss — a formal hardware verification system: User's guide," University of Brtish Columbia Department of Computer Science, Tech. Rep. TR-93-45, Dec. 1993.

[20] J. Joyce and C.-J. Seger, "Linking BDD-based symbolic evaluation to interactive theorem-proving," in *Design Automation Conference*, 1993, pp. 469–474.

[21] C.-J. Seger and J. Joyce, "A mathematically precise two-level formal hardware verification methodology," Department of Computer Science, University of British Columbia, Report 92-34, Dec. 1992.

[22] S. Hazelhurst and C.-J. Seger, "A simple theorem prover based on symbolic trajectory evaluation and BDD's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 4, pp. 413–422, 1995.

[23] M. Kaufmann and J. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[24] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[25] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *ACM/IEEE Design Automation Conference*, 1998, pp. 538–541.

[26] ——, "Formal verification using parametric representations of boolean constraints," in *ACM/IEEE Design Automation Conference*, 1999, pp. 402–407.

[27] R. Milner, "The use of machines to assist in rigorous proof," in *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*. Prentice-Hall, 1985, pp. 77–88.

[28] J. Harrison, "HOL light: A tutorial introduction," in *Proc. Formal Methods in Computer-Aided Design (FMCAD'96)*, ser. LNCS, M. Srivas and A. Camilleri, Eds., vol. 1166. Springer-Verlag, 1996, pp. 265–269.

[29] Coq Development Team, *The Coq Proof Assistant: Reference Manual, V8.4*. Inria, Aug. 2012.

[30] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer-Verlag, 2002, vol. 2283.

[31] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation," in *Design Automation Conference*. ACM Press, Jun. 1997, pp. 167–172.

[32] T. Melham, *Higher Order Logic and Hardware Verification*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993, vol. 31.

[33] *Edinburgh LCF: A Mechanised Logic of Computation*, ser. LNCS. Springer-Verlag, 1979, vol. 78.

[34] Z. Khasidashvili, G. Gavrielov, and T. Melham, "Assume-guarantee validation for STE properties within an SVA environment," in *Formal Methods in Computer-Aided Design: FMCAD 2009*. IEEE, 2009, pp. 108–115.

[35] S. Rajan, N. Shankar, and M. Srivas, "An integration of model-checking with automated proof checking," in *Computer-Aided Verification*, ser. LNCS, vol. 939. Springer-Verlag, Jun. 1995, pp. 84–97.

[36] J. Sawada and E. Reeber, "ACL2SIX : A hint used to integrate a theorem prover and an automated verification tool," in *Proceedings of Formal Methods in Computer Aided Design: FMCAD 2006*. IEEE Computer Society, 2006, pp. 161–170.

[37] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham, "The PROSPER toolkit," *Int. J. on Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 189–210, Feb. 2003.

[38] N. Shankar, "Symbolic analysis of transition systems," in *Abstract State Machines: Theory and Applications*, ser. LNCS, no. 1912. Springer-Verlag, 2000, pp. 287–302.

[39] M. Kaufmann and J. S. Moore, *ACL2 and Its Applications to Digital System Verification*. Springer-Verlag, 2010, pp. 1–21.

[40] W. A. Hunt, Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, Dec. 1989.

[41] N. Een. ABC/ZZ. [Online]. Available: https://bitbucket.org/niklaseen/abc-zz

[42] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. [Online]. Available: http://www.eecs.berkeley.edu/ alanmi/abc/

[43] W. A. Hunt, Jr. and S. Swords, "Centaur technology media unit verification," in *Computer Aided Verification*, ser. LNCS, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 353–367.

[44] D. A. Jamsek, *Symbolic Trajectory Evaluation*, ch. 12, pp. 185–200.

[45] S. O. Swords, "A verified framework for symbolic execution in the ACL2 theorem prover," Ph.D. dissertation, University of Texas as Austin, 2010. [Online]. Available: http://hdl.handle.net/2152/ETD-UT-2010-12-2210

[46] T. F. Melham and R. B. Jones, "Abstraction by symbolic indexing transformations," in *Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 2517. Springer-Verlag, 2002, pp. 1–18.

Satisfiability Modulo ODEs

Sicun Gao
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
Email: sicung@cs.cmu.edu

Soonho Kong
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
Email: soonhok@cs.cmu.edu

Edmund M. Clarke
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
Email: emc@cs.cmu.edu

***Abstract*—We study SMT problems over the reals containing ordinary differential equations,. They are important for formal verification of realistic hybrid systems and embedded software. We develop δ-complete algorithms for SMT formulas that are purely existentially quantified, as well as $\exists\forall$-formulas whose universal quantification is restricted to the time variables. We demonstrate scalability of the algorithms, as implemented in our open-source solver dReal, on SMT benchmarks with several hundred nonlinear ODEs and variables.**

I. Introduction

Hybrid systems tightly combine finite automata and continuous dynamics. In most cases, the continuous components are specified by ordinary differential equations (ODEs). Thus, formal verification of general hybrid systems requires reasoning about logic formulas over the reals that contain ODE constraints. This problem is considered very difficult and has not been investigated in the context of decision procedures until recently [7], [8], [16]. It is believed that current techniques are not powerful enough to handle formulas that arise from formal verification of realistic hybrid systems, which typically contain many nonlinear ODEs and other constraints.

Since the first-order theory over the reals with trigonometric functions is already undecidable, solving formulas with general ODEs seems inherently impossible. We have resolved much of this theoretical difficulty by proposing the study of δ-*complete* decision procedures for such formulas [10]. An algorithm is δ-complete for a set of SMT formulas, where δ is an arbitrary positive rational number, if it correctly decides whether a formula is unsatisfiable or δ-satisfiable. Here, a formula is δ-satisfiable if, under some δ-*perturbations*, a syntactic variant of the original formula is satisfiable [9]. We have shown that δ-complete decision procedures are suitable for various formal verification tasks [9], [10]. We have also proved that δ-complete decision procedures exist for SMT problems over the reals with Lipschitz-continuous ODEs. Such results serve as a theoretical foundation for developing practical decision procedures for the SMT problem.

In this paper we study practical δ-complete algorithms for SMT formulas over the reals with ODEs. We show that such algorithms can be made powerful enough to scale to realistic benchmark formulas with several hundred nonlinear ODEs.

We develop decision procedures for the problem following a standard DPLL(ICP) framework, which relies on constraint solving algorithms as studied in Interval Constraint Propagation (ICP) [2]. In this framework, for any ODE system we can consider its solution function $\vec{x}_t = \vec{f}(t, \vec{x}_0)$ as a constraint between the initial variables $\vec{x}_0$, time variable t, and the final state variables $\vec{x}_t$. We define pruning operators that take interval assignments on $\vec{x}_0$, t, and $\vec{x}_t$ as inputs, and output refined interval assignments on these variables. We formally prove that the proposed algorithms are δ-complete. Beyond standard SMT problems where all variables are existentially quantified, we also study $\exists\forall$-formulas under the restriction that the universal quantifications are limited to the time variables (we call them $\exists\forall^t$-formulas). Such formulas have been an obstacle in SMT-based verification of hybrid systems [4], [5].

In brief, this paper makes the following contributions:

- We formalize the SMT problem over the reals with general Lipschitz-contiunous ODEs, and illustrate its expressiveness by encoding various standard problems concerning ODEs: initial and boundary value problems, parameter synthesis problems, differential algebraic equations, and bounded model checking of hybrid systems. In some cases, $\exists\forall^t$-formulas are needed.

- We propose algorithms for solving SMT with ODEs, using ODE constraints to design pruning operators in a branch-and-prune framework. We handle both standard SMT problems with only existentially quantified variables, as well as $\exists\forall^t$-formulas. We prove that the algorithms are δ-complete.

- We demonstrate the scalability of the algorithms, as implemented in our open-source solver dReal [11], on realistic benchmarks encoding formal verification problems for several nonlinear hybrid systems.

Related Work. Solving real constraints with ODEs has a wide range of applications, and much previous work exists for classes with special structures in different paradigms [6], [13], [18]. Recently [12] proposed a more general constraint solving framework, focusing on the formulation of the problem in the standard CP framework. On the SMT solving side, several authors have considered logical combinations of ODE constraints and proposed partial decision procedures [7], [8], [16]. We aim to extend and formalize existing algorithms for a general SMT theory with ODES, and formally prove that they can be made δ-complete. In terms of practical performance, the proposed algorithms are made scalable to various benchmarks that contain hundreds of nonlinear ODEs and variables.

The paper is organized as follows. In Section II, we define the SMT problem with ODEs and show how it can encode various standard problems with ODEs. In Section III, we propose algorithms in the DPLL(ICP) framework for solving fully existentially quantified formulas as well as $\exists\forall^t$ formulas. In Section IV we show experimental results.

ISBN 978-0-9835678-3-7/13.

II. SMT over the Reals with ODEs

A. A First-Order Signature with Computable Real Functions

As studied in Computable Analysis [19], [17], we can encode real numbers as infinite strings, and develop a computability theory of real functions using Turing machines that perform operations using oracles encoding real numbers. We briefly review definitions and results of importance to us. Throughout the paper we use $||\cdot||$ to denote the max norm $||\cdot||_\infty$ over $\mathbb{R}^n$ for various n. First, a *name* of a real number is a sequence of rational numbers converging to it:

Definition 1 (Names). *A name of $a \in \mathbb{R}$ is any function $\gamma_a : \mathbb{N} \to \mathbb{Q}$ that satisfies: for any $i \in \mathbb{N}$, $|\gamma_a(i) - a| < 2^{-i}$. For $\vec{a} \in \mathbb{R}^n$, $\gamma_{\vec{a}}(i) = \langle \gamma_{a_1}(i), ..., \gamma_{a_n}(i)\rangle$. We write the set of all possible names for $\vec{a}$ as $\Gamma(\vec{a})$.*

Next, a real function f is *computable* if there is a Turing machine that can use any argument x of f as an oracle, and compute the value of $f(x)$ up to an arbitrary precision 2^{-i}, where $i \in \mathbb{N}$. Formally:

Definition 2 (Computable Functions). *We say $f :\subseteq \mathbb{R}^n \to \mathbb{R}$ is computable if there exists an oracle Turing machine $\mathcal{M}_f$ such that for any $\vec{x} \in dom(f)$, any name $\gamma_{\vec{x}}$ of $\vec{x}$, and any $i \in \mathbb{N}$, the machine uses $\gamma_{\vec{x}}$ as an oracle and i as an input to compute a rational number $M_f^{\gamma_{\vec{x}}}(i)$ satisfying $|M_f^{\gamma_{\vec{x}}}(i) - f(\vec{x})| < 2^{-i}$.*

The definition requires that for any $\vec{x} \in \text{dom}(f)$, with access to an arbitrary oracle encoding the name $\gamma_{\vec{x}}$ of $\vec{x}$, M_f outputs a 2^{-i}-approximation of $f(\vec{x})$. In other words, the sequence $M_f^{\gamma_{\vec{x}}}(1), M_f^{\gamma_{\vec{x}}}(2), ...$ is a name of $f(\vec{x})$. Intuitively, f is computable if an arbitrarily good approximation of $f(\vec{x})$ can be obtained using any good enough approximation to any $\vec{x} \in \text{dom}(f)$. Most common continuous real functions are computable [19]. Addition, multiplication, absolute value, $\min$, $\max$, $\exp$, $\sin$. Compositions of computable functions are computable. In particular, solution functions of Lipschitz-continuous ordinary differential equations are computable, as we explain next.

B. Solution Functions of ODEs

We now show that the framework of computable functions allows us to consider solution functions of ODE systems.

Notation 3. *We use $\vec{x} = \vec{y}$ between n-dimensional vectors to denote the system of equations $x_i = y_i$ for $1 \le i \le n$.*

Let $D \subseteq \mathbb{R}^n$ be compact and $g_i : D \to \mathbb{R}$ be n Lipschitz-continuous functions, which means that for some constant $c_i \in \mathbb{R}^+$ $(1 \le i \le n)$, for all $\vec{x}_1, \vec{x}_2 \in D$,

$$|g_i(\vec{x}_1) - g_i(\vec{x}_2)| \le c_i||\vec{x}_1 - \vec{x}_2||.$$

Let t be a variable over $\mathbb{R}$. We consider the first-order autonomous ODE system

$$\frac{d\vec{y}}{dt} = \vec{g}(\vec{y}(t, \vec{x}_0)) \text{ and } \vec{y}(0, \vec{x}_0) = \vec{x}_0 \tag{1}$$

where $\vec{x}_0 \in D$. Here, each

$$y_i : \mathbb{R} \times D \to \mathbb{R} \tag{2}$$

is called the i-th solution function of the ODE system (1). A key result in computable analysis is that these solution functions are computable, in the sense of Definition 2:

Proposition 4 ([17]). *The solution functions $\vec{y}$ in the form of (2) of the ODE system (1) are computable over $\mathbb{R} \times D$.*

To see why this is true, recall that for any $t \in \mathbb{R}$ and $\vec{x}_0 \in D$, the value of the solution function follows the Picard-Lindelöf form:

$$\vec{y}(t, \vec{x}_0) = \int_0^t \vec{g}(\vec{y}(s, \vec{x}_0))ds + \vec{x}_0.$$

Approximations of the right-hand side of the equation can be computed by finite sums, theoretically up to an arbitrary precision.

C. SMT Problems and δ-Complete Decision Procedures

We now let $\mathcal{F}$ denote an arbitrary collection of computable real functions, which can naturally contain solution functions of ODE systems in the form of (2). Let $\mathcal{L}_{\mathcal{F}}$ denote the first-order signature $\langle \mathcal{F}, < \rangle$, where constants are seen as 0-ary functions in $\mathcal{F}$. Let $\mathbb{R}_{\mathcal{F}}$ be the structure $\langle \mathbb{R}, \mathcal{F}^{\mathbb{R}}, <^{\mathbb{R}} \rangle$ that interprets $\mathcal{L}_{\mathbb{R}_{\mathcal{F}}}$-formulas in the standard way. We focus on formulas whose variables take values from bounded domains, which can be defined using bounded quantifiers:

Definition 5 (Bounded Quantifiers). *The bounded quantifiers $\exists^{[u,v]}$ and $\forall^{[u,v]}$ are defined as*

$$\exists^{[u,v]} x.\varphi =_{df} \exists x.(u \le x \wedge x \le v \wedge \varphi),$$
$$\forall^{[u,v]} x.\varphi =_{df} \forall x.((u \le x \wedge x \le v) \to \varphi),$$

where u and v denote $\mathcal{L}_{\mathbb{R}_{\mathcal{F}}}$ terms, whose variables only contain free variables in φ excluding x. It is easy to check that $\exists^{[u,v]} x.\varphi \leftrightarrow \neg\forall^{[u,v]} x.\neg\varphi$.

The key definition in our framework is δ-*variants* of first-order formulas:

Definition 6 (δ-Variants). *Let $\delta \in \mathbb{Q}^+ \cup \{0\}$, and φ a bounded $\mathcal{L}_{\mathbb{R}_{\mathcal{F}}}$-sentence of the standard form*

$$\varphi : \ Q_1^{I_1} x_1 \cdots Q_n^{I_n} x_n \ \psi[t_i(\vec{x}) > 0; t_j(\vec{x}) \ge 0],$$

where $i \in \{1, ...k\}$ and $j \in \{k+1, ..., m\}$. Note that negations are represented by sign changes on the terms. The δ-weakening φ^δ of φ is defined as the result of replacing each atom $t_i > 0$ by $t_i > -\delta$ and $t_j \ge 0$ by $t_j \ge -\delta$. That is,

$$\varphi^{-\delta} : \ Q_1^{I_1} x_1 \cdots Q_n^{I_n} x_n \ \psi[t_i(\vec{x}) > -\delta; t_j(\vec{x}) \ge -\delta].$$

The SMT problem is standardly defined as deciding satisfiability of quantifier-free formulas, which is equivalent to deciding the truth value of fully existentially quantified sentences. We will also consider formulas that are partially universally quantified. Thus, we consider both Σ_1 and Σ_2 formulas here.

Definition 7 (Bounded Σ_1- and Σ_2-SMT Problems). *A Σ_1-SMT problem is a formula of the form*

$$\exists^{I_1} x_1 \cdots \exists^{I_n} x_n.\varphi(\vec{x})$$

and a Σ_2-SMT problem is of the form

$$\exists^{I_1} x_1 \cdots \exists^{I_n} x_n \forall^{I_{n+1}} x_{n+1} \cdots \forall^{I_m} x_m.\varphi(\vec{x}).$$

In both cases $\varphi(\vec{x})$ is a quantifier-free $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula.

Definition 8 (δ-Completeness [9])**.** *Let S be a set of $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$ formulas, and $\delta \in \mathbb{Q}^+$. We say a decision procedure A is δ-complete for S, if for any $\varphi \in S$, A correctly returns one of the following answers*

- *φ is false;*
- *$\varphi^{-\delta}$ is true.*

If the two cases overlap, either one is correct.

We have proved in [10] that δ-complete decision procedures exists for arbitrary bounded $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-sentences. In particular, there exists δ-complete decision procedures for the bounded Σ_1 and Σ_2 SMT problems. This serves as the theoretical foundation as well as a correctness requirement for the practical algorithms that we will develop in the following sections.

D. SMT Encoding of Standard Problems with ODEs

In this section, we list several standard problems related to ODE systems and show that they can be easily encoded and generalized through SMT formulas. They motivate the development of decision procedures for the theory.

Remark 9. *In all the following cases, solutions to the standard problems are obtained from witnesses for the existentially quantified variables in the SMT formulas.*

Remark 10. *In the definitions below, when the solution functions $\vec{y}$ of ODE systems are written as part of a formula, no analytic forms are needed. They are functions included in the signature $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$.*

Generalized Initial Value Problems. Given an ODE system, the standard initial value problem asks for a solution of the variables at certain time, given a complete assignment to the initial conditions of the system. In the form of SMT formulas, we easily allow the initial conditions to be constrained by arbitrary quantifier-free $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formulas:

Definition 11 (Generalized IVP)**.** *Let $X \subseteq \mathbb{R}^n$ be a compact domain, $T \in \mathbb{R}^+$, and $\vec{y} : [0,T] \times X \to X$ be the computable solution functions of an ODE system. Let $t \in [0,T]$ be an arbitrary constant that represents a time point of interest. The generalized IVP problem is defined by formulas of the form:*

$$\exists^X x_0 \exists^X \vec{x}.\ \varphi(\vec{x}_0) \wedge \vec{x} = \vec{y}(t, \vec{x}_0),$$

where φ is a quantifier-free $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula constraining the initial states $\vec{x}_0$, and $\vec{x}$ is the needed value for time point t.

Generalized Boundary Value Problems. Given an ODE system, the standard boundary value problem is concerned with computing the computable solution function when part of the variables are assigned values at the beginning of flow, and part of the variables as assigned values at the end of the flow. A generalized version as encoded by SMT formulas is:

Definition 12 (Generalized BVP)**.** *Let $X \subseteq \mathbb{R}^n$ be a compact domain, $T \in \mathbb{R}^+$, and $\vec{y} : [0,T] \times X \to X$ be the solution functions of an ODE system. Let $t, t' \in [0,T]$ be two time points of interest. The generalized BVP problem is:*

$$\exists^X x_0 \exists^X \vec{x}_t \exists^X \vec{x}.\varphi(\vec{x}_0, \vec{x}_t, t) \wedge \vec{x}_t = \vec{y}(t, \vec{x}_0) \wedge \vec{x} = \vec{y}(t', \vec{x}_0)$$

where φ is a quantifier-free $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula that specifies the boundary conditions. Note that $\vec{x}$ is the value that we are interested in solving in the chosen time point t'.

Data-Fitting and Parameter Synthesis. The data fitting problem is the following. Suppose an ODE system has part of its parameters unspecified. Given a sequence of data $(t_1, \vec{a}_1), ..., (t_k, \vec{a}_k)$, we need to find the values of the missing parameters of the original ODE system. More formally:

Definition 13 (Data-Fitting Problems)**.** *Let $X \subseteq \mathbb{R}^n$ and $P \subseteq \mathbb{R}^m$ be compact domains, $T \in \mathbb{R}^+$, and $\vec{y}(\vec{p}) : [0,T] \times X \to X$ be the solution functions of an ODE system, where $\vec{p} \in P$ be a vector of parameters. Let $(t_1, \vec{a}_1), ..., (t_k, \vec{a}_k)$ be a sequence of pairs in $[0,T] \times X$. The data-fitting problem is defined by:*

$$\exists^P \vec{p} \exists^X x_0.\ \varphi(\vec{x}_0) \wedge \vec{a}_1 = \vec{y}(\vec{p}, t_1, \vec{x}_0) \wedge \cdots \wedge \vec{a}_k = \vec{y}(\vec{p}, t_k, \vec{x}_0),$$

where a quantifier-free φ constraints the initial states $\vec{x}_0$.

Differential Algebraic Equations. DAE problems combine ODEs and algebraic constraints:

$$\frac{d\vec{y}}{dt} = \vec{g}(\vec{y}(t, \vec{y}_0), \vec{z}) \quad (3)$$

$$0 = \vec{h}(\vec{y}, \vec{z}, t) \quad (4)$$

where $\vec{y}, \vec{y}_0 \in \mathbb{R}^n$, $\vec{z} \in \mathbb{R}^m$. To express the problem in $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$, we need to use extra universal quantification to ensure that the algebraic relations hold throughout the time duration. Again, we can also generalize the equation in (4) to an arbitrary quantifier-free $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$-formula. The problem is encoded as:

Definition 14 (DAE Problems)**.** *Let $X \subseteq \mathbb{R}^n$ be a compact domain, $T \in \mathbb{R}^+$, and $\vec{y} : [0,T] \times X \times X \to X$ be the computable solution functions of the ODE system in (3) parameterized by $\vec{z}$. Let h be defined by (4). Let $t \in [0,T]$ be a time point of interest. A DAE problem is defined by the following formula:*

$$\exists^X \vec{x}_0 \exists^X \vec{x} \exists^Z \vec{z} \forall^{[0,t]} t'.$$
$$\varphi(\vec{x}_0) \wedge \vec{x} = \vec{y}(t, \vec{x}_0, \vec{z}) \wedge h(\vec{y}(\vec{x}_0, t'), \vec{z}, t') = 0$$

where a quantifier-free φ specifies the initial conditions for $\vec{y}$, and $\vec{x}$ is the needed value at time point t.

Bounded Model Checking of Hybrid Systems. Bounded model checking problems for hybrid systems can be naturally encoded as SMT formulas with ODEs [7], [8], [16], [4], [5]. We consider a simple hybrid system to show an example. Let H be an n-dimensional 2-mode hybrid system. In mode 1, the flow of the system follows an ODE system whose solution function is $\vec{y}_1(t, \vec{x}_0)$, and in mode 2, it follows another solution function $\vec{y}_2(t, \vec{x}_0)$. The jump condition from mode 1 to mode 2 is specified by $\mathsf{jump}(\vec{x}, \vec{x}')$. The invariants are specified by $\mathsf{inv}_i(\vec{x})$ and for mode i. Let $\mathsf{unsafe}(\vec{x})$ denote an unsafe region. Let the continuous variables be bounded in X and time be bounded in $[0,T]$. Now, if H starts from mode 1 with initial states satisfying $\mathsf{init}(\vec{x})$, it can reach the unsafe region after one discrete jump from mode 1 to mode 2, iff the following formula is true:

$$\exists^X \vec{x}_1 \exists^X \vec{x}_1^t\ \exists^X \vec{x}_2 \exists^X \vec{x}_2^t\ \exists^{[0,T]} t_1 \exists^{[0,T]} t_2\ \forall^{[0,t_1]} t_1' \forall^{[0,t_2]} t_2'.$$
$$\mathsf{init}(\vec{x}_1) \wedge \vec{x}_1^t = \vec{y}_1(t_1, \vec{x}_1) \wedge \mathsf{inv}_1(\vec{y}_1(t_1', \vec{x}_1)) \wedge \mathsf{jump}(\vec{x}_1^t, \vec{x}_2)$$
$$\wedge\ \vec{x}_2^t = \vec{y}_2(t_2, \vec{x}_2) \wedge \mathsf{inv}_2(\vec{y}_2(t_2', \vec{x}_2)) \wedge \mathsf{unsafe}(\vec{x}_2^t).$$

The encoding can be explained as follows. For each mode, we use two variable vectors $\vec{x}_i$ and $\vec{x}_i^t$ to represent the continous flows. $\vec{x}_i$ denote the starting values of a flow, and $\vec{x}_i^t$ denotes the final values. In mode 1, the flow starts with some values in the initial states, specified by $\mathsf{init}(\vec{x}_1)$. Then, we follow the continuous dynamics in mode 1, so that $\vec{x}_1^t$ denotes the final value $\vec{x}_1^t = \vec{y}(t_1, \vec{x}_1)$. Then the system follows the jumping condition and resets the variables from $\vec{x}_1^t$ to $\vec{x}_2$ as specified by $\mathsf{jump}(\vec{x}_1^t, \vec{x}_2)$. After that, the system follows the flow in mode 2. In the end, we check if the final state $\vec{x}_2^t$ in mode 2 satisfies the unsafe predicate, $\mathsf{unsafe}(\vec{x}_2)$.

III. Algorithms

A. The ICP framework

The method of Interval Constraint Propagation (ICP) [2] finds solutions of real constraints using a "branch-and-prune" method that performs constraint propagation of interval assignments on real variables. The intervals are represented by floating-point end-points. Only over-approximations of the function values are used, which are defined by interval extensions of real functions.

Definition 15 (Floating-Point Intervals and Hulls)**.** *Let $\mathbb{F}$ denote the finite set of all floating point numbers with symbols $-\infty$ and $+\infty$ under the conventional order $<$. Let*

$$\mathbb{IF} = \{[a,b] \subseteq \mathbb{R} : a, b \in \mathbb{F}, a \leq b\} \text{ and } \mathbb{BF} = \bigcup_{n=1}^{\infty} \mathbb{IF}^n$$

denote the set of closed real intervals with floating-point end-points, and the set of boxes *with these intervals, respectively. When $S \subseteq \mathbb{R}^n$ is a set of real numbers, the hull of S is:*

$$\mathrm{Hull}(S) = \bigcap \{B \in \mathbb{BF} : S \subseteq B\}.$$

Definition 16 (Interval Extension [2])**.** *Suppose $f :\subseteq \mathbb{R}^n \to \mathbb{R}$ is a real function. An interval extension operator $\natural(\cdot)$ maps f to a function $\natural f :\subseteq \mathbb{BF} \to \mathbb{IF}$, such that for any $B \in dom(\natural f)$, it is always true that $\{f(\vec{x}) : \vec{x} \in B\} \subseteq \natural f(B)$.*

Algorithm 1 ICP($f_1, ..., f_m, B_0 = I_1^0 \times \cdots \times I_n^0, \delta$)

```
1:  S ← B_0
2:  while S ≠ ∅ do
3:      B ← S.pop()
4:      while ∃1 ≤ i ≤ m, B ≠_δ Prune(B, f_i) do
5:          B ← Prune(B, f_i)
6:      end while
7:      if B ≠ ∅ then
8:          if ∃1 ≤ i ≤ n, |♮f_i(B)| ≥ δ then
9:              {B_1, B_2} ← Branch(B, i)
10:             S.push({B_1, B_2})
11:         else
12:             return sat
13:         end if
14:     end if
15: end while
16: return unsat
```

ICP uses interval extensions of functions to "prune" out sets of points that are not in the solution set, and "branch" on intervals when such pruning can not be done, until a small enough box that may contain a solution is found. A high-level description of the decision version of ICP is given in Algorithm 1. In Algorithm 1, $\mathrm{Branch}(B, i)$ is an operator that returns two smaller boxes $B' = I_1 \times \cdots \times I_i' \times \cdots \times I_n$ and $B'' = I_1 \times \cdots \times I_i'' \times \cdots \times I_n$, where $I_i \subseteq I_i' \cup I_i''$. The key component of the algorithm is the $\mathrm{Prune}(B, f)$ operation. Any operation that contracts the intervals on variables can be seen as pruning, but for correctness we need formal requirements on the pruning operator in ICP. Basically, we need to require that the interval extensions of the functions converge to the true values of the functions, and that the pruning operations are well-defined, as specified below.

Definition 17 (δ-Regular Interval Extensions)**.** *We say an interval extension $\natural f$ of $f : \mathbb{R}^n \to \mathbb{R}$ is δ-regular, if for some constant $c \in \mathbb{R}$, for any $B \in \mathbb{R}^n$, $|\natural f(B)| \leq \max(c||B||, \delta)$.*

Definition 18 (Well-defined Pruning Operators [9])**.** *Let $\mathcal{F}$ be a collection of real functions, and $\natural$ be a δ-regular interval extension operator on $\mathcal{F}$. A* well-defined (equality) pruning operator *with respect to $\natural$ is a partial function $\mathrm{Prune}_\natural :\subseteq \mathbb{BF} \times \mathcal{F} \to \mathbb{BF}$, such that for any $f \in \mathcal{F}$, $B, B' \in \mathbb{BF}$,*

1) $\mathrm{Prune}_\natural(B, f) \subseteq B$;
2) *If* $\mathrm{Prune}_\natural(B, f) \neq \emptyset$, *then* $0 \in \natural f(\mathrm{Prune}_\natural(B, f))$;
3) $B \cap \{\vec{a} \in \mathbb{R}^n : f(\vec{a}) = 0\} \subseteq \mathrm{Prune}_\natural(B, f)$.

When $\natural$ is clear, we simply write Prune. The rules can be explained as follows. (W1) ensures that the algorithm always makes progress. (W2) ensures that the result of a pruning is always a reasonable box that may contain a zero, and otherwise B is pruned out. (W3) ensures that the real solutions are never discarded. We proved the following theorem in [9]:

Theorem 19. *Algorithm 1 is δ-complete if the pruning operators are well-defined.*

B. ODE Pruning in an ICP Framework

We now study the algorithms for SMT formulas with ODEs. The key is to design the appropriate pruning operators for the solution functions of ODE systems. The pruning operations here strengthen and formalize the ones proposed in [7], [8], [12], such that δ-completeness can be proved.

We recall some notations first. Let $D \subseteq \mathbb{R}^n$ be compact and $g_i : D \to D$ be n Lipschitz-continuous functions. Given the first-order autonomous ODE system

$$\frac{d\vec{y}}{dt} = \vec{g}(\vec{y}(t, \vec{x}_0)) \text{ and } \vec{y}(0, \vec{x}_0) = \vec{x}_0 \tag{5}$$

where $\vec{x}_0 \in D$, we write

$$y_i : [0, T] \times D \to D_i$$

to represent the i-th solution function of the ODE system. The δ-regular interval extension of y_i is an interval function

$$\natural y_i : (\mathbb{IF} \cap [0, T]) \times (\mathbb{BF} \cap D) \to \mathbb{IF}$$

such that for a constant $c \in \mathbb{R}$, for any time domain $I_t \subseteq \mathbb{IF} \cap [0, T]$ and any box of initial values $B_{\vec{x}_0} \subseteq \mathbb{BF} \cap D$, we have

$$\{x_t \in \mathbb{R} : x_t = y_i(t, \vec{x}_0), \vec{x}_0 \in B_{\vec{x}_0}, t \in I_t\} \subseteq \natural y_i(I_t, B_{\vec{x}_0})$$

and

$$|\sharp y_i(I_t, B_{\vec{x}_0})| \leq \max(c \cdot ||I_t \times B_{\vec{x}_0}||, \delta).$$

We will also need the notion of the *reverse* of the ODE system (5), as defined by

$$\frac{d\vec{y}_-}{dt} = \vec{g}_-(\vec{y}_-(t, \vec{x}_t)) \text{ and } \vec{y}(0, \vec{x}_t) = \vec{x}_t. \tag{6}$$

Here, $\vec{g}_-$ is defined as $-\vec{g}$, the vector of functions consisting of the negation of each function in $\vec{g}$, which is equivalent to reversing time in the flow defined by the ODE system. That is, for $\vec{x}_0, \vec{x}_t \in D, t \in \mathbb{R}$, we always have

$$\vec{x}_t = \vec{y}(t, \vec{x}_0) \text{ iff } \vec{x}_0 = \vec{y}_-(t, \vec{x}_t). \tag{7}$$

Naturally, we write $\sharp(y_-)_i$ to denote the δ-regular interval extension of the i-th component of $\vec{y}_-$.

Algorithm 2 ODEPruning$(\sharp\vec{y}, B_{\vec{x_0}}, B_{\vec{x_t}}, I_t)$

repeat
  $B'_{\vec{x_t}} \leftarrow \text{Prune}_{\text{fwd}}(\sharp\vec{y}, B_{\vec{x_0}}, B_{\vec{x_t}}, I_t)$
  $I'_t \leftarrow \text{Prune}_{\text{time}}(\sharp\vec{y}, B_{\vec{x_0}}, B'_{\vec{x_t}}, I_t)$
  $B'_{\vec{x_0}} \leftarrow \text{Prune}_{\text{bwd}}(\sharp\vec{y}, B_{\vec{x_0}}, B'_{\vec{x_t}}, I'_t)$
until $B_{\vec{x_0}} = B'_{\vec{x_0}} \wedge B_{\vec{x_t}} = B'_{\vec{x_t}} \wedge I_t = I'_t$
return $(B'_{\vec{x_0}}, B'_{\vec{x_t}}, I'_t)$

The relation between the initial variables $\vec{x}_0$, the time duration t, and the flow variables $\vec{x}_t$ is specified by the constraint $\vec{x}_t = \vec{y}(t, \vec{x}_0)$. Given the interval assignment on any two of $\vec{x}_0$, $\vec{x}_t$, and t, we can use the constraint to obtain a refined interval assignment to the third variable vector. Thus, we can define three pruning operators as follows.

Remark 20. *The precise definitions of the pruning operators should map the interval assigments on all variables to new assignments on all variables. For notational simplicity, in the pruning operators below we only list the assignments that are actually changed between inputs and outputs. For instance, the forward pruning operator only changes the values on $B_{\vec{x}_t}$.*

Forward Pruning. Given interval assignments on $\vec{x}_0$ and t, we compute a refinement of the interval assignments on $\vec{x}_t$. Figure 1 depicts the forward pruning operation. Formally, we define the following operator:

Definition 21 (Forward Pruning)**.** *Let $\vec{y} : [0, T] \times D \to D$ be the solution functions of an ODE system. Let $B_{\vec{x}_0}$, $B_{\vec{x}_t}$, and I_t be interval assignments on the variables $\vec{x}_0$, $\vec{x}_t$, and t. We define the forward-pruning operator as:*

$$\text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y}) = \text{Hull}\left(B_{\vec{x}_t} \cap \sharp\vec{y}(I_t, B_{\vec{x}_0})\right).$$

Backward Pruning. Given interval assignments on $\vec{x}_t$ and t, we can compute a refinement of the interval assignments on $\vec{x}_0$ using the reverse of the solution function. Figure 2 depicts backward pruning. Formally, we define the following operator:

Definition 22 (Backward Pruning)**.** *Let $\vec{y} : [0, T] \times D \to D$ be the solution functions of an ODE system, and let $\vec{y}_-$ be the reverse of $\vec{y}$. Let $B_{\vec{x}_0}$, $B_{\vec{x}_t}$, and I_t be interval assignments on the variables $\vec{x}_0$, $\vec{x}_t$, and t. We define the backward-pruning operator as:*

$$\text{Prune}_{\text{bwd}}(B_{\vec{x}_0}, \vec{y}) = \text{Hull}\left(B_{\vec{x}_0} \cap \sharp\vec{y}_-(I_t, B_{\vec{x}_t})\right).$$

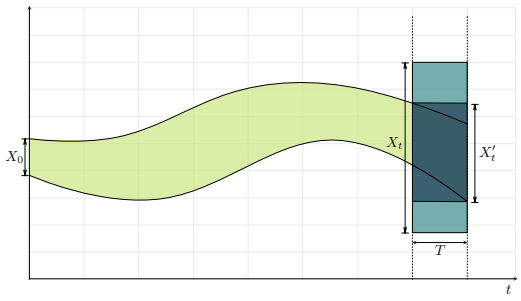


Fig. 1: Forward Pruning. X_0, X_t, t represents the current interval assignments, and X'_t is the refined interval assignment on $\vec{x}_t$ after pruning.

Algorithm 3 $\text{Prune}_{\text{fwd}}(\sharp\vec{y}, B_{\vec{x_0}}, B_{\vec{x_t}}, I_t)$

$B'_{\vec{x_t}} \leftarrow \phi$
$I_{\Delta t} \leftarrow [I_t^l, I_t^l + \varepsilon]$
while $I_{\Delta t}^u < I_t^u$ **do**
  $B'_{\vec{x_t}} \leftarrow \text{Hull}(B'_{\vec{x_t}} \cup \sharp\vec{y}(I_{\Delta t}, B_{\vec{x_0}}))$
  $I_{\Delta t} \leftarrow I_{\Delta t} + \varepsilon$
end while
return $B_{\vec{x_t}} \cap B'_{\vec{x_t}}$

Time-Domain Pruning. Given interval assignments on $\vec{x}_0$ and $\vec{x}_t$, we can also refine the interval assignment on t by pruning out the time intervals that do not contain any $\vec{x}_t$ that is consistent with the current interval assignments on $\vec{x}_t$. Figure 3 depicts time-domain pruning. Formally, we define the following operator:

Definition 23 (Time-Domain Pruning)**.** *Let $\vec{y} : [0, T] \times D \to D$ be the solution functions of an ODE system. Let $B_{\vec{x}_0}$, $B_{\vec{x}_t}$, I_t be interval assignments on the variables $\vec{x}_0$, $\vec{x}_t$, and t. We define the time-domain pruning operator as:*

$$\text{Prune}_{\text{time}}(I_t, \vec{y}) = \text{Hull}\left(I_t \cap \{I : \sharp\vec{y}(I, B_{\vec{x}_0}) \cap B_{\vec{x}_t} \neq \emptyset\}\right).$$

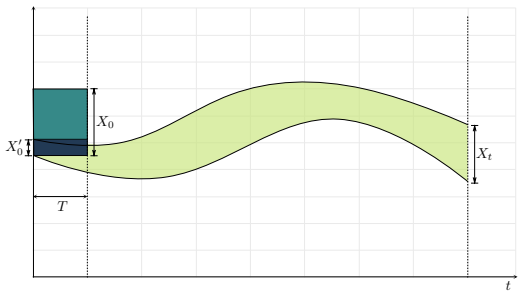


Fig. 2: Backward Pruning. X_0, X_t, t represents the current interval assignments, and X'_0 is the refined interval assignment on $\vec{x}_0$ after pruning.

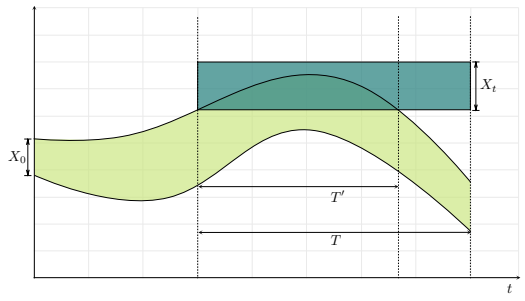


Fig. 3: Time-Domain Pruning. X_0, X_t, t represents the current interval assignments, and T' is the refined interval assignment on t after pruning.

Overall, the pruning algorithm on based on ODE constraints iteratively applies the three pruning operators until a fixed point on the interval assignments is reached.

Algorithm 4 $\text{Prune}_{\text{bwd}}(\sharp\vec{y}, B_{\vec{x_0}}, B_{\vec{x_t}}, I_t)$

$B'_{\vec{x_0}} \leftarrow \phi$
$I_{\Delta t} \leftarrow [I_t^l, I_t^l + \varepsilon]$
while $I_{\Delta t}^u < I_t^u$ **do**
  $B'_{\vec{x_0}} \leftarrow \text{Hull}(B'_{\vec{x_0}} \cup \sharp\vec{y}_-(I_{\Delta t}, B_{\vec{x_t}}))$
  $I_{\Delta t} \leftarrow I_{\Delta t} + \varepsilon$
end while
return $B_{\vec{x_0}} \cap B'_{\vec{x_0}}$

We show the more detailed steps in the three pruning operations in Algorithm 2, 3, 4, and 5.

Algorithm 5 $\text{Prune}_{\text{time}}(\sharp\vec{y}, B_{\vec{x_0}}, B_{\vec{x_t}}, I_t)$

$I'_t \leftarrow \phi$
$I_{\Delta t} \leftarrow [I_t^l, I_t^l + \varepsilon]$
while $I_{\Delta t}^u < I_t^u$ **do**
  $B'_{\vec{x_t}} \leftarrow \sharp\vec{y}(I_{\Delta t}, B_{\vec{x_0}})$
  if $B'_{\vec{x_t}} \cap B_{\vec{x_t}} \neq \phi$ **then**
    $I'_t = \text{Hull}(I'_t \cup I_{\Delta t})$
  else
    $I_{\Delta t} \leftarrow I_{\Delta t} + \varepsilon$
  end if
end while
return I'_t

Theorem 24. *The three pruning operators are well-defined.*

Proof: We prove that the forward pruning operator is well-defined, and the proofs for the other two operators are similar. Note that the definitions of well-defined pruning are formulated for equality constraints compared to 0. Here we use the function $f = \vec{y}(t, \vec{x}_0) - \vec{x}_t$ in the pruning operator. (Strictly speaking f is a function vector that evaluates to $\vec{0}$ on points satisfying the ODE flow. Here for notational simplicity we just write f as a single-valued function and compare with the scalar 0.)

First, (W1) is satisfied because of the simple fact that for any boxes $B_1, B_2 \in \mathbb{BF}$, we have $\text{Hull}(B_1 \cap B_2) \subseteq B_1$.

Next, suppose $0 \notin \sharp f(\text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y}) - B_{\vec{x}_t})$. Then there does not exist any $\vec{a}_t \in \mathbb{R}^n$ that satisfies both $\vec{a}_t \in B_{\vec{x}_t}$ and $\vec{a}_t \in \text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y})$. Since at the same time

$$\text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y}) = \text{Hull}\left(B_{\vec{x}_t} \cap \sharp\vec{y}(I_t, B_{\vec{x}_0})\right) \subseteq B_{\vec{x}_t},$$

this requires that $\text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y}) = \emptyset$. Consequently (W2) is satisfied.

Third, note that $\sharp\vec{y}(I_t, B_{\vec{x}_0})$ is an interval extension of $\vec{y}$. Thus, for any $\vec{a}_t \in \mathbb{R}^n$ such that $\vec{y}(t, \vec{x}_0)$ for some $t \in I_t$ and $\vec{x}_0 \in B_{\vec{x}_0}$, we have $\vec{a}_t \in \sharp\vec{y}(I_t, B_{\vec{x}_0})$. Following the definition of the pruning operator, we have $\vec{a}_t \in \text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y})$. Thus, $B_{\vec{x}_t} \cap Z_f \subseteq \text{Prune}_{\text{fwd}}(B_{\vec{x}_t}, f)$ and (W3) holds. ■

C. $\exists\forall^t$-Formulas and Low-Order Approximations

For $\exists\forall$-formulas, if the universal quantification is only over the time variables, we can follow the trajectory and prune away the assignment on $\vec{x}_0$, $\vec{x}_t$, and t that violates the constraints on the universally quantified time variable. In fact, although the extra quantification complicates the problem, the universal constraints improve the power of the pruning operations.

Here we focus on problems with one ODE system, which can be easily generalized. Let $\vec{y}$ denote the solution functions of an ODE system, we consider an $\exists\forall^t$-formula of the form

$$\exists^X \vec{x}_0 \exists^X \vec{x}_t \exists^{[0,T]} t \forall^{[0,t]} t'.\ \vec{x}_t = \vec{y}(t, \vec{x}_0) \wedge \varphi(\vec{y}(t', \vec{x}_0)) \qquad (8)$$

Note that the problems encoded as Σ_2-SMT formulas as listed in Section II-D are all of this form.

We consider $\varphi(\vec{y}(t', \vec{x}_0))$ as a special constraint on the $\vec{x}_0$ and t variables. Using this constraint, we can further refine the three pruning operators as follows.

Definition 25 (Pruning Refined by $\forall^t$-Constraints)**.** *Let $\vec{y} : [0, T] \times D \to \mathbb{R}^n$ be the solution functions of an ODE system. Let $B_{\vec{x}_0}$, $B_{\vec{x}_t}$, and I_t be interval assignments on the variables $\vec{x}_0$, $\vec{x}_t$, and t. Let $\varphi(\vec{y}(t', \vec{x}_0))$ be a constraint on the universally quantified time variable, as in (8). We first define*

$$\sharp\varphi(I_t, B_{\vec{x}_0}) = \text{Hull}(\{\vec{a} \in \mathbb{R}^n : \vec{a} = \vec{y}(t, \vec{x}_0), t \in I_t, \vec{x}_0 \in B_{\vec{x}_0},\ \textit{and } \varphi(\vec{a}) \textit{ is true.}\})$$

and define $\sharp\varphi_-$ by replacing $\vec{y}$ with $\vec{y}_-$ in the definition above. The forward pruning operator with φ, written as $\text{Prune}^{\varphi}_{\text{fwd}}(B_{\vec{x}_t}, \vec{y})$, is defined as

$$\text{Hull}\left(B_{\vec{x}_t} \cap \sharp\vec{y}(I_t, B_{\vec{x}_0}) \cap \sharp\varphi(I_t, B_{\vec{x}_0})\right)$$

Backward pruning $\text{Prune}^{\varphi}_{\text{bwd}}(B_{\vec{x}_0}, \vec{y})$ is defined as

$$\text{Hull}\left(B_{\vec{x}_0} \cap \sharp\vec{y}_-(I_t, B_{\vec{x}_t}) \cap \sharp\varphi_-(I_t, B_{\vec{x}_t})\right).$$

Time-domain pruning $\text{Prune}^{\varphi}_{\text{time}}(I_t, \vec{y})$ is defined as

$$\text{Hull}\left(I_t \cap \{I : \sharp\vec{y}(I, B_{\vec{x}_0}) \cap B_{\vec{x}_t} \cap \sharp\varphi(I_t, B_{\vec{x}_0}) \neq \emptyset\}\right).$$

In general, $\sharp\varphi$ can be computed by a recursive call to DPLL(ICP), by solving the Σ_1-SMT problem $\varphi(\vec{x})$. In many practical applications, φ is of some simple form such as $\vec{a} \leq \vec{x}_t \leq \vec{b}$, in which case simple pruning is shown in Figure 4. Another useful heuristic in ODE pruning is to bound the range

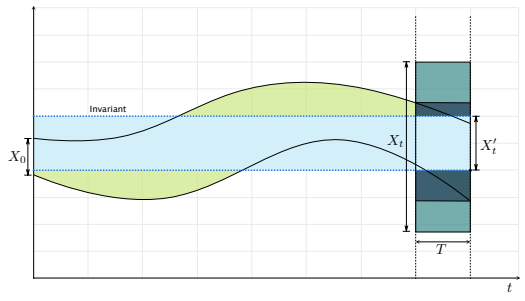


Fig. 4: Pruning with $\forall^t$-Constraints

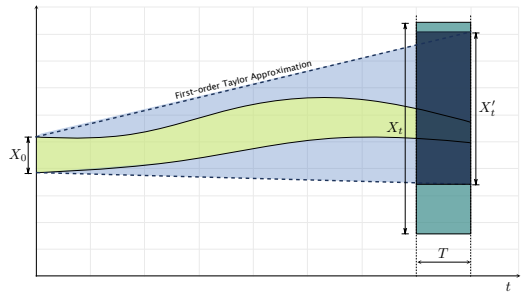


Fig. 5: Pruning with Low-Order Taylor Approximations

of the derivatives for a vector space specified by $\vec{g}$. Suppose for any time $t \in [0, T]$, the derivatives $\vec{g}$ are bounded in $[\vec{l}_g, \vec{u}_g]$. Then by the Picard-Lindelöf representation, we have

$$\vec{x}_t = \int_0^t \vec{g}(\vec{y}(s, \vec{y}_0))ds + \vec{y}_0 \in [0, T] \cdot [\vec{l}_g, \vec{u}_g] + B_{\vec{x}_0}$$

We can use this formula to perform preliminary pruning on $\vec{x}_t$, which is especially efficient when combined with $\forall^t$-constraints. Figure 5 illustrates this pruning method.

IV. Experiments

Our tool dReal implements the procedures we studied for solving SMT formulas with ODEs. It is built on several existing packages, including opensmt [3] for the general DPLL(T) framework, realpaver [14] for ICP, and CAPD [1] for computing interval-enclosures of ODEs. The tool is open-source at http://dreal.cs.cmu.edu. All benchmarks and data shown here are also available on the tool website.

All experiments were conducted on a machine with a 3.4GHz octa-core Intel Core i7-2600 processor and 16GB RAM, running 64-bit Ubuntu 12.04LTS. Table I is a summary of the running time of the tool on various SMT formulas generated from bounded model checking hybrid systems. The formulas typically contain a large number of variables and nonlinear ODEs.

The AF model as we show in Table I is obtained from [15]. It is a precise model of atrial fibrillation, a serious cardiac disorder. The continuous dynamics in the model concerns four

P	#M	#D	#O	#V	delta	R	Time(s)	Trace
AF	4	3	20	44	0.001	S	43.10	90K
AF	8	7	40	88	0.001	S	698.86	20M
AF	8	23	120	246	0.001	S	4528.13	59M
AF	8	31	160	352	0.001	S	8485.99	78M
AF	8	47	240	528	0.001	S	15740.41	117M
AF	8	55	280	616	0.001	S	19989.59	137M
CT	2	2	15	36	0.005	S	345.84	3.1M
CT	2	2	15	36	0.002	S	362.84	3.1M
EO	3	2	18	42	0.01	S	52.93	998K
EO	3	2	18	42	0.001	S	57.67	847K
EO	3	11	72	168	0.01	U	7.75	–
BB	2	10	22	66	0.01	S	0.25	123K
BB	2	20	42	126	0.01	S	0.57	171K
BB	2	20	42	126	0.001	S	2.21	168K
BB	2	40	82	246	0.01	U	0.27	—-
BB	2	40	82	246	0.001	U	0.26	—-
D1	3	2	9	24	0.1	S	30.84	72K
DU	3	2	6	16	0.1	U	0.04	–

TABLE I: #M = Number of modes in the hybrid system, #D = Unrolling depth, #O = Number of ODEs in the unrolled formula, #V = Number of variables in the unrolled formula, R = Bounded Model Checking Result (delta-SAT/UNSAT) Time = CPU time (s), Trace = Size of the ODE trajectory, AF = Atrial Filbrillation, CT = Cancer Treatment, EO = Electronic Oscillator, BB = Bouncing Ball with Drag, D1,DU = Decay Model.

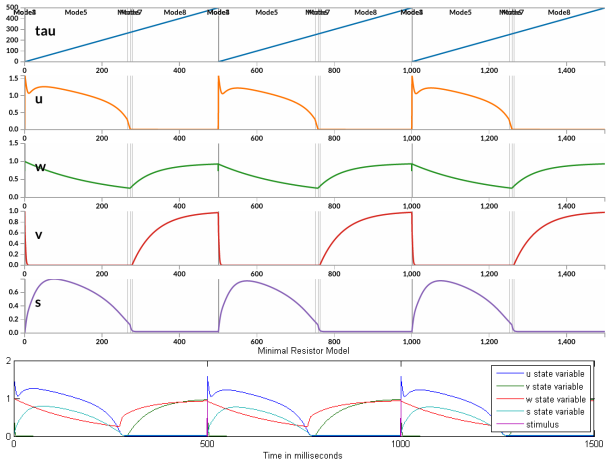


Fig. 6: Above: Witness for the AF model at depth 23 and 1500 time units. Below: Experimental simulation data.

state variables and the ODEs are highly nonlinear, such as:

$$\begin{aligned}
\frac{du}{dt} &= e + (u - \theta_v)(u_u - u)vg_{fi} + wsg_{si} - g_{so}(u) \\
\frac{ds}{dt} &= \frac{g_{s2}}{(1 + e^{-2k(u-us)})} - g_{s2}s \\
\frac{dv}{dt} &= -g_v^+ \cdot v \qquad \frac{dw}{dt} = -g_w^+ \cdot w
\end{aligned}$$

The exponential term on the right-hand side of the ODE is the sigmoid function, which often appears in modelling biological switches. On this model, our tool is able to perform a depth-55 unrolling, and solve the generated logic formula. Such a formula contains 280 nonlinear ODEs of the type shown here,

with 616 variables. The computed trace from dReal suggests a witness of the reachability property that can be confirmed by experimental simulation. Figure 6 shows the comparison between the trace computed from bounded model checking and the actual experimental simulation trace.

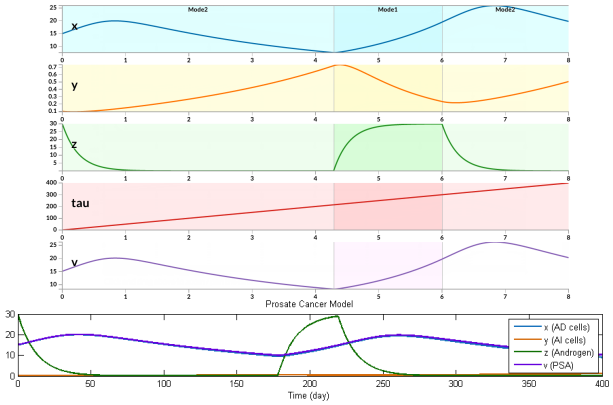


Fig. 7: Above: Witness computed for the CT model at depth 3 and 500 time units. Below: Experimental simulation data.

The CT model represents a prostate cancer treatment model that contains nonlinear ODEs such as following:

$$\begin{aligned}
\frac{dx}{dt} &= (\alpha_x(k_1 + (1-k_1)\frac{z}{z+k_2} \\
&\quad -\beta_x((1-k_3)\frac{z}{z+k_4} + k_3)) - m_1(1-\frac{z}{z_0}))x + c_1 x \\
\frac{dy}{dt} &= m_1(1-\frac{z}{z_0})x + (\alpha_y(1-d\frac{z}{z_0}) - \beta_y)y + c_2 y \\
\frac{dz}{dt} &= \frac{-z}{\tau} + c_3 z \\
\frac{dv}{dt} &= (\alpha_x(k_1 + (1-k_1)\frac{z}{z+k_2} - \beta_x(k_3 + (1-k_3)\frac{z}{z+k_4})) \\
&\quad -m_1(1-\frac{z}{z_0}))x + c_1 x + m_1(1-\frac{z}{z_0})x \\
&\quad +(\alpha_y(1-d\frac{z}{z_0}) - \beta_y)y + c_2 y
\end{aligned}$$

The EO model represents an electronic oscillator model that contains nonlinear ODEs such as the following:

$$\begin{aligned}
\frac{dx}{dt} &= -ax \cdot sin(\omega_1 \cdot \tau) \\
\frac{dy}{dt} &= -ay \cdot sin((\omega_1 + c_1) \cdot \tau) \cdot sin(\omega_2) \cdot 2 \\
\frac{dz}{dt} &= -az \cdot sin((\omega_2 + c_2) \cdot \tau) \cdot cos(\omega_1) \cdot 2 \\
\frac{\omega_1}{dt} &= -c_3 \cdot \omega_1 \quad \frac{\omega_2}{dt} = -c_4 \cdot \omega_2 \quad \frac{d\tau}{dt} = 1
\end{aligned}$$

The other models are standard simple nonlinear models (for instance, bouncing ball with nonlinear friction), on which our tool has no difficulty in solving.

V. Conclusion

In this paper we have studied SMT problems over the real numbers with ODE constraints. We have developed δ-complete algorithms in the DPLL(ICP) framework, for both the standard SMT formulas that are purely existentially quantified, as well as $\exists\forall$-formulas whose universal quantification is restricted to the time variables. We have demonstrated the scalability of our approach on nonlinear SMT benchmarks. We believe that the proposed decision procedures can scale on nonlinear problems and can serve as the underlying engine for formal verification of realistic hybrid systems and embedded software.

Ackowledgements. We are grateful for many important suggestions from Jeremy Avigad, Andreas Eggers, and Martin Fränzle. In particular, we formulated the notion of δ-regular interval extensions to avoid technical difficulties that Eggers and Fränzle pointed out to us. We thank the anonymous referees for various important comments.

References

[1] CAPD: Computer assisted proofs in dynamical systems. http://capd.ii.uj.edu.pl/index.php.

[2] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16. Elsevier, 2006.

[3] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The opensmt solver. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer, 2010.

[4] A. Cimatti, S. Mover, and S. Tonetta. A quantifier-free SMT encoding of non-linear hybrid automata. In *FMCAD*, pages 187–195, 2012.

[5] A. Cimatti, S. Mover, and S. Tonetta. SMT-based verification of hybrid systems. In *AAAI*, 2012.

[6] J. Cruz and P. Barahona. Constraint satisfaction differential problems. In *CP*, pages 259–273, 2003.

[7] A. Eggers, M. Fränzle, and C. Herde. SAT modulo ODE: A direct SAT approach to hybrid systems. In *ATVA*, pages 171–185, 2008.

[8] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle. Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In G. Barthe, A. Pardo, and G. Schneider, editors, *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2011.

[9] S. Gao, J. Avigad, and E. M. Clarke. Delta-complete decision procedures for satisfiability over the reals. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2012.

[10] S. Gao, J. Avigad, and E. M. Clarke. Delta-decidability over the reals. In *LICS*, pages 305–314, 2012.

[11] S. Gao, S. Kong, and E. Clarke. dReal: An SMT solver for nonlinear theories over the reals. *CADE*, 2013.

[12] A. Goldsztejn, O. Mullier, D. Eveillard, and H. Hosobe. Including ordinary differential equations based constraints in the standard cp framework. In D. Cohen, editor, *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2010.

[13] L. Granvilliers. Parameter estimation using interval computations. *SIAM J. Sci. Comput.*, 26(2):591–612, Feb. 2005.

[14] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.

[15] R. Grosu, G. Batt, F. H. Fenton, J. Glimm, C. L. Guernic, S. A. Smolka, and E. Bartocci. From cardiac cells to genetic regulatory networks. In *CAV*, pages 396–411, 2011.

[16] D. Ishii, K. Ueda, and H. Hosobe. An interval-based sat modulo ode solver for model checking nonlinear hybrid systems. *STTT*, 13(5):449–461, 2011.

[17] K.-I. Ko. *Complexity Theory of Real Functions*. BirkHauser, 1991.

[18] Y. Lin and M. A. Stadtherr. Guaranteed state and parameter estimation for nonlinear continuous-time systems with bounded-error measurements. *Industrial and Engineering Chemistry Research*, pages 7198–7207, 2007.

[19] K. Weihrauch. *Computable Analysis: An Introduction*. 2000.

Verifying Global Convergence for a Digital Phase-Locked Loop

Jijie Wei Yan Peng Ge Yu Mark Greenstreet
University of British Columbia

***Abstract*—We present a verification of a digital phase-locked loop (PLL) using the SpaceEx hybrid-systems tool. In particular, we establish global convergence – from any initial state the PLL eventually reaches a state of phase and frequency lock. Having shown that the PLL converges to a small region, traditional methods of circuit analysis based on linear-systems theory can be used to characterize the response of the PLL when in lock. The majority of the verification involves modeling each component of the PLL with piece-wise linear differential inclusions. We show how non-linear transfer functions, quantization error, and other non-idealities can be included in such a model. A limitation of piece-wise linear inclusions is that the linear coefficients for each component must take on fixed values. For real designs, ranges will be specified for these components. We show how a key step of the verification can be generalized to handle interval values for the linear coefficients by using an SMT solver.**

***Index Terms*—analog/mixed-signal verification, digital PLL, global stability, hybrid systems, linear differential inclusions, non-linear systems, SMT.**

I. Introduction

Phase-locked loops (PLLs) are ubiquitous in analog and mixed-signal designs. Their uses include frequency multiplication to generate a high-frequency clock from a lower frequency reference, for clock-acquisition in high-speed links, and as modulators and demodulators for wireless communication. Recently, PLL design has shifted from traditional, analog, charge-pump based designs to various "all-digital" architectures. Several consequences of device scaling to smaller feature sizes have motivated this transition including: greater device-to-device parameter variation impairs designs that depend on matched circuits; lower power supply voltages removes the "voltage headroom" needed for high-quality, on-chip current sources; and the scaling of passive components such as inductors and capacitors lags that for transistors. A failed PLL can block further test of an entire chip or major subsystem; thus, there is a high value in verifying correctness of PLL designs. This paper presents the formal verification of global convergence for the digital PLL published in [1].

Functional verification of analog blocks such as PLLs can be divided into two parts: global convergence to an intended operating point, and small-signal analysis at the operating point. The key insight here is that nearly all analog blocks are *intended* to have some kind of linear response when at or near their intended operating point [2]. Existing analysis techniques such as periodic AC analysis (PAC) [3] are available in standard commercial CAD tools such as Spectre® from Cadence. These techniques allow designers to characterize key performance properties of analog blocks such as the jitter, power-supply sensitivity, and tracking bandwidth of a PLL. A designer can have high confidence in the correct functioning of their block *assuming* that it reaches its intended operating point.

To show that an analog block will reach its intended operating point from any initial state is the *global convergence* problem. Here, the non-linearities of the circuit must be taken into account. Simulation based methods are impractical both because of the impossibility of covering all possible inputs, initial states, and operating conditions, and because individual simulations may need to cover thousands of cycles of the PLL's oscillator to show the locking behaviour. To support power-management techniques such as dynamic voltage-frequency scaling and power down modes, PLLs may need to start-up or change lock frequency tens to hundreds of times per second. If a PLL occasionally fails to lock, then the chip is useless. Tracking down such bugs on real silicon can be extremely difficult. Thus, proving global convergence is of great value for real designs.

In this paper, a digital PLL is modeled as a piecewise-linear hybrid-automaton, and global convergence of the automaton is shown. The first step of this approach is to formulate hybrid-automaton models for each component of the PLL. We note that the basic components of oscillators, dividers, phase comparators, and integrators are common to all PLL designs; thus, we expect this work to be re-usable for verifying other PLL structures. We use the SpaceEx [4] tool to show that all initial states converge to a small region around the intended operating point. To obtain practical verification times, we found it necessary to identify "phases" that the PLL passes through when converging to its operating point. By structuring our hybrid automaton model to reflect these phases, we avoid SpaceEx needing to perform costly fix point computations.

A limitation of the SpaceEx based approach is that the model parameters of the piece-wise linear inclusions are fixed. This requires giving specific values to some analog quantitities that a designer can only guarantee to be in some range. We show that for the PLL from [1], a global Lyapunov function (i.e. progress function) can be constructed using basic methods from linear systems theory. We then use the Z3 SMT solver [5] to show global convergence for a simplified model of the DPLL.

The key contributions of this paper are:

- We verify global convergence for a digital PLL. Another approach to digital PLL verification that was developed independently was recently reported in [6]. We believe

ISBN 978-0-9835678-3-7/13.

that these are the first such verifications for digital PLLs.
- We show how each component of the digital PLL can be modeled as a hybrid automaton. Our models account for non-linearities of the components, quantization, and other non-idealities.
- We demonstrate how convergence can be shown by reachability analysis (using SpaceEx) and by solving systems on non-linear inequalities (using Z3).

II. Related Work

There have been several previously published reports of PLL verification using formal methods. The earliest verification that we know of was by [7, chap. 6]. Dhingra's design uses a fixed-frequency oscillator that is intended to operate at N times the frequency of a reference. The PLL adaptively chooses edges of its internal oscillator to approximate the edges of the lower frequency reference. The time resolution is limited by the period of the internal oscillator. While this may be useful for low frequency applications such as audio frequency modems, we are not aware of any such PLLs in use for the more standard PLL applications of clock generation, clock-data-recovery, and wireless communication. Dhingra verified the tracking behaviour of his design using the HOL theorem prover.

More recently, Dong *et al.* [8], [9] proposed using property checking for AMS verification, including PLLs. They used "symbolic recurrence equations" as a property specification language, and show how this can be used to automatically construct a monitor to check simulation runs to see if a PLL locks in the required time for that run. This does not address the problems of long-simulation times to show that a PLL locks or the incompleteness of simulation based approaches to show convergence.

Shortly after the work by Dong *et al.*, Jesser and Hedrich [10] described a model-checking result for an analog PLL with an XOR-gate phase detector. They performed symbolic model checking using MTBDDs to represent both the discrete and analog parts. They state that the four-dimensional analog state space is partitioned into 2^{11} hyperboxes, and that next-box relations are determined by random simulation. It is not clear how they guarantee the complete coverage with this approach.

Two years ago, Althoff *et al.* [11] presented the verification of a charge-pump PLL using an approach that they refer to as "continuization." They use a purely linear model for the components of their PLL, and their focus is on the switching activities of the phase-frequency detector, in particular, uncertainties in switching delays. Their approach also verifies the PLL for ranges of component parameters. We present an SMT-based technique for handling interval parameters in Section V. Althoff *et al.* is the only other work that we are aware of that accounts for such variation.

More recently, Lin et al. [6] independently developed an approach for verifying a digital PLL using SMT techniques. To the best of our knowledge, they are the first to claim formal verification of a digital PLL. Similar to the approach presented in this paper, they consider a purely linear, analog model and then reason about the discrepancies between this idealized model and a digital implementation. They use the KRR SMT solver to verify bounds on this discrepancy. They verify bounds on the lock time of a digitally intensive PLL assuming that most of the digital variables are initialized to fixed values, and that only the oscillator phase is unknown. Our work shows initialization for a different PLL design over the complete state space.

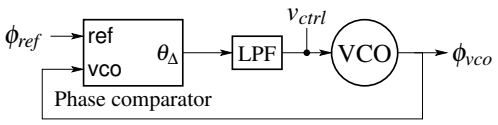


Fig. 1. A Simple PLL

III. The Digital PLL

A. A PLL Primer

The function of a phase-locked loop (PLL) is to adjust the PLL's oscillator so that it tracks the frequency and phase of a reference signal. Figure 1 shows a simple PLL. The PLL sets the control voltage, v_{ctrl} of the VCO according to the phase difference between the VCO and the reference and the integral of this difference to match the VCO's frequency to that of the reference and align their phases. Simply, if the PLL's oscillator lags the reference, then v_{ctrl} increases; and the VCO frequency increases so that the VCO will catch up with the reference. Conversely, if the PLL's oscillator is ahead of the reference, then v_{ctrl} will decrease causing the PLL's oscillator frequency to decrease.

In more detail, the voltage-controlled oscillator (VCO) can be understood as a voltage-to-frequency converter. Phase is the integral of frequency; so we can express the phase of the VCO output as $\theta_{vco} = (\int v_{ctrl} dt) \bmod [-\pi, +\pi)$. Phase is modular, and we write $\theta \bmod [-\pi, +\pi)$ to indicate the value in $[-\pi, +\pi)$ that is congruent with θ modulo 2π radians. The phase comparator generates an output voltage that is proportional to the phase difference of its inputs: $\theta_\Delta = (\theta_{vco} - \theta_{ref}) \bmod [-\pi, +\pi]$. The reference is assumed to have a constant frequency, ω_{ref}; thus $\theta_{ref} = \omega_{ref} t$, where t is time. The low pass filter implements the integral and proportional correction terms with $v_{ctrl} = a_0 \theta_\Delta + a_1 \int \theta_\Delta dt$. Combining these equations and differentiating twice, we get:

$$\theta_{vco} \quad = \quad a_0 \int \theta_\Delta dt + a_1 \int \int \theta_\Delta dt\, dt \qquad (1)$$

In the simple case where $a_0 = 0$ and $a_1 = 0$, the PLL becomes a simple harmonic oscillator. The frequency of the PLL oscillates with mean value of ω_{ref} but never converges. If both a_0 and a_1 are negative, then there is a unique solution where the PLL's oscillator converges to the frequency and phase of the reference. Note that if all of the components are linear, then simple algebraic techniques suffice to show (or refute) global convergence.

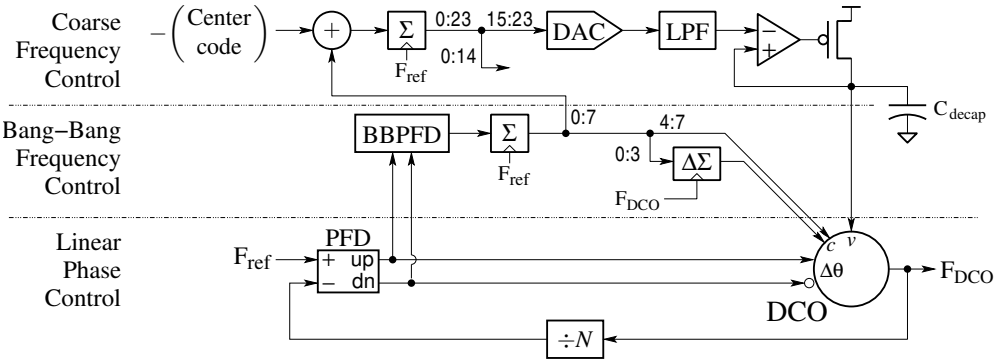
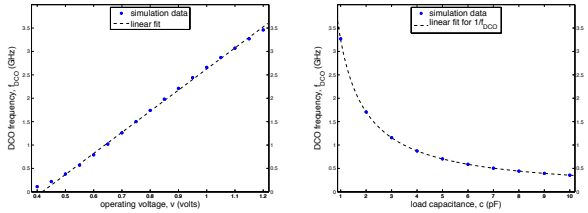


Fig. 2. The digital PLL from [1]

For real designs, the components are not perfectly linear. The component may very closely match their linear idealizations when the PLL has locked to the reference, but significant non-linearities may occur when out of lock. Furthermore, the analog components of the simple PLL are difficult to implement in advanced fabrication technologies. For example, large capacitors are needed to implement the integrator part of the low pass filter. For these and other reasons, designers are making more and more extensive use of digital and mixed signal designs for PLLs.



(a) frequency vs. operating voltage

(b) frequency vs. load capacitance

Fig. 3. Ring-oscillator response

B. The Digital PLL

Figure 2 shows the digital PLL architecture from [1]. While this real-world PLL has many more components than the simple PLL from Fig. 1, its operation is similar. The digitally controlled oscillator (DCO) performs the role of the VCO from the simple PLL. The divider, $\div N$ is added to make the lock-point for the DCO a multiple of the reference frequency. The phase comparator of the simple PLL is replaced by two phase-frequency detectors – a linear PFD that reports the phase difference of the reference and the frequency divided DCO, and a "bang-bang" PFD that only reports the sign of this phase difference. The remaining components implement the low-pass filter of the simple PLL with the accumulators functioning as integrators.

The three control paths of the PLL (linear phase, bang-bang frequency, and coarse frequency) work together to set the frequency, f_{dco}, of the digitally controlled oscillator (DCO) to N times the reference frequency, f_{ref}, and to align the phase of the DCO and the reference (i.e., rising edges of the reference clock should coincide with rising edges of the DCO).

The digitally-controlled oscillator in [1] is a three-stage ring-oscillator with three control inputs: v, c, and θ_Δ. The v input sets the operating voltage of the DCO. First-order transistor models suggest that f_{dco} should be roughly proportional to v. Figure 3(a) shows the results of Spectre® simulations of a ring-oscillator in a 65nm CMOS process with a 1.2V nominal V_{dd}. For an operating voltage v with $0.5 \leq v \leq 1.2$, a linear fit provides a good approximation of the DCO frequency.

The c input controls switches that add capacitors to increase the load for each stage of the ring oscillator. As seen in Fig. 3(b), the oscillator period (inverse frequency) is very accurately modeled as a linear function of load capacitance. In addition to the four, binary weighted, values from c, a fifth bit with weight equal to the least significant bit, is provided from the Delta-Sigma modulator. This provides a period averaging to improve the frequency resolution of the bang-bang control path.

Note that the DCO frequency, f_{dco} is proportional to a linear function of v and *inversely proportional* to a linear function of c. We can write this as

$$f_{dco} = \frac{1+\alpha v}{1+\beta c} f_0 \qquad (2)$$

where f_0 is the frequency of the DCO (extrapolated) to where v and c are at zero, and α and β are the sensitivities to v and c respectively. Most importantly, when modeling the response to both v and c, either their ranges must be quite small, or the model is inherently non-linear. To show global convergence, the non-linearity of the DCO must be included in the model.

The θ_Δ input controls a linear phase path. Each stage of the ring-oscillator consists of an inverter in parallel with two tri-state inverters. One of the tri-state inverters is *enabled* when up is asserted, and the other is *disabled* when dn is asserted. This causes the oscillator to run faster when up is asserted without dn, and slower when dn is asserted without up. This produces a phase shift advance (resp. delay) to the length of time that the oscillator is in its fast (resp. slow) mode.

The frequency divider, labeled $\div N$ in figure 2 is a modulo-N counter. Its output has a period N times that of the DCO.

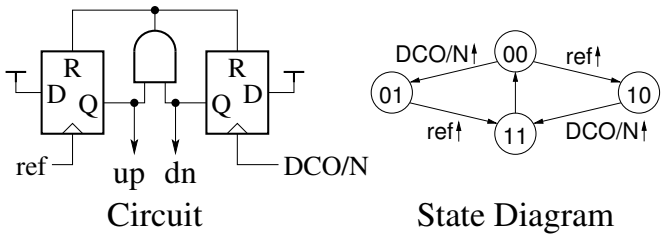


Fig. 4. The PFD

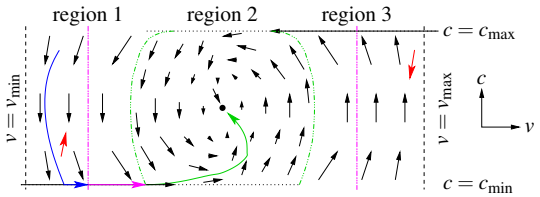


Fig. 5. Operation with saturating integrators based on linear model from Eq. 3, but not to scale.

The two phase-frequency detectors (PFD and BBPFD) in the diagram determine if the DCO should increase or decrease its frequency. Figure 4 shows a typical circuit for the linear phase-frequency detector, labeled PFD in Fig. 2, and its state transition diagram. Each state is labeled with the value of the up and dn (down) outputs of the PFD – for example, in state 01 the up signal is low and the dn signal is high. The PFD is reset to state 00 after it has received rising edges from both the reference clock, ref, and the frequency divided DCO, DCO/N. If the next event is a rising edge from the divided DCO, this indicates that the DCO is ahead of the reference in phase or running at a higher frequency. Accordingly, the PFD enters state 01 (asserting dn) indicating that the DCO should decrease in frequency (equivalently, drop back in phase). Likewise, if the first event after resetting is a rising edge of the reference, then up is asserted. Note that the *duration* of asserting dn without up (or vice-versa) indicates the phase difference between the reference and DCO. This is the principle behind the linear-phase control path of the PLL.

The BBPFD is a so-called "bang-bang" phase frequency detector. It simply detects whether the linear PFD asserts up before dn or dn before up and then outputs $+1$ or -1 respectively. The output of the BBPFD indicates the direction that the DCO should move, but does not contain the magnitude information of the linear PFD.

The accumulators implement integrators. The integrator in the bang-bang path directly controls the c input of the DCO. The Delta-Sigma modulator provides an averaging: if the lower four bits of the accumulator for c encode a value of k, then the output of the Delta-Sigma modulator will be asserted for k out of 16 cycles of the DCO. The integrator in the coarse frequency control path is designed to have low-bandwidth to ensure low jitter (cycle-to-cycle variation of the DCO period). The output of the integrator is converted to a voltage with the digital-to-analog converter (DAC). To suppress the coupling of power-supply noise into the DCO, an additional low-pass filter and linear regulator are included in the loop.

To understand the operation of the digital PLL when $\frac{1}{N} f_{dco} \neq f_{ref}$, the difference will be noted by the PFD and the BBPFD. The BBPFD will output a value to drive c in the direction to correct for the frequency difference. For small frequency differences, this bang-bang loop provides fast tracking. For larger differences, the accumulator for c will saturate at its minimum or maximum value. Any of these situations lead to c being different from the center code. This drives the coarse frequency loop to change v in the direction needed to bring $\frac{1}{N} f_{dco}$ to f_{ref} and to return c to the center code value. Because the DCO has only a fixed set of oscillation frequencies, for most values of f_{ref} there is no choice of c and v that causes $\frac{1}{N} f_{dco}$ to be *exactly* f_{ref}. This leads to limit-cycle oscillations. The linear phase control path suppresses these oscillations.

IV. Verifying Convergence with SpaceEx

We divided our verification effort into two main tasks: designer and verifier. One of us (the "designer") wrote Matlab models of the components (such as the DCO, phase-frequency detectors, etc.) and used these for simulations. The other person was the "verifier" who translated these models into hybrid automata and used SpaceEx to verify global convergence. We found it very helpful to start with a simple, completely linear model that reflected the structure of digital PLL and gradually refine it and add details to faithfully model the actual PLL.

A. Modeling the digital PLL

A simple linear model. Let

$$\begin{array}{rcl} f_{dco} & = & \alpha v - \beta c \\ \frac{d}{dt}\theta_\Delta & = & \frac{1}{N} f_{dco} - f_{ref} - g_\theta \theta_\Delta \\ \frac{d}{dt} c & = & -g_1 \theta_\Delta \\ \frac{d}{dt} v & = & g_2 (c - c_{center}) \end{array} \tag{3}$$

where α, β, N, and θ_Δ are as described in Section III; g_θ is the "time gain' for the linear phase path; g_1 is the integrator gain for c; g_2 is the integrator gain of the v path. Because the model is linear, global convergence can be shown (or refuted) by simple, analytical methods. If $g_\theta > 0$, $g_1 < 0$, and $g_2 < 0$, then this linear PLL converges globally.

Modeling the accumulators. The accumulators of the digital PLL approximate the integrators of the linear model by computing a Reimann sum on rounded values of the integrand. The accumulators add perturbations due to quantization and saturation. Saturation of the accumulators is modeled by providing bounds for c and v: $c_{\min} \leq c \leq c_{\max}$ and $v_{\min} \leq v \leq v_{\max}$. The model is based on Eq. 3 with the change that if c reaches $c_{\min}$ and $\dot{c} < 0$ by Eq. 3, then $\dot{c} = 0$ in this "saturating" model. Likewise for the cases when $c = c_{\max}$ or when v reaches one of its bounds.

Simulations of the Matlab model with saturating integrators showed the behaviour depicted in Fig. 5. The colored path shows a typical trajectory, and the red arrows show an artifact that is can be caused by the internal delays of the PFD that will be discussed later in this section. In region 1, v is too low to achieve $\frac{1}{N} f_{dco} = f_{ref}$. In this case, c reaches its saturation value of $c_{\min}$ (the blue curved path), and then v increases (the blue and magenta arrow)until $\frac{1}{N} f_{dco} \approx f_{ref}$. At this point $\dot{c} > 0$ and the trajectory enters region 2. Trajectories in region 2 asymptotically approach the equilibrium point (the curved green path) without further saturation of c or v. In region 3, v is too high, and c saturates at $c_{\min}$ until the trajectory enters region 2. The corresponding hybrid automaton has seven modes: four for saturated values of c or v, and one for each of the regions described above. Again, SpaceEx readily showed global convergence.

The observation that the phase locked loop first saturates c, then gets v close to its final value, and the converges in both c and v applies to the actual PLL as well as to this simplified model. This observation allowed us to describe the PLL in a way where SpaceEx shows the convergence of one variable at a time. In the course of verifying the PLL, cycles in the mode-transition graph would cause time-outs while SpaceEx tried to compute fix points. The "one variable at a time" approach eliminated the most egregious of these cycles from the model and achieved very practical execution times.

The SpaceEx model approximates the values of the accumulators of the digital PLL with integrators. Thus, the error analysis is basically that for a Riemann sum approximation of an integral, but in this case the integral is approximating the sum rather than the other way around. Let $f : \mathbb{R}^+ \to \mathbb{R}$ be an integrable function such that for all $t \geq 0$. $|f(t)| \leq F$ and $\left|\frac{d}{dt} f(t)\right| \leq D$ for some $F, D \in \mathbb{R}^+$. Let $\text{round}(x, \gamma)$ denote the rounding of x to the nearest integer multiple of γ, for any $\gamma \geq 0$. Then for $\Delta T > 0$

$$\Delta T \sum_{k=0}^{N-1} \text{round}(f(k\Delta T), \gamma) \;=\; \int_0^{N\Delta T} f(u) + \mu(u)\, du + \rho(t) \qquad (4)$$

for some $\mu, \rho : \mathbb{R}^+ \to \mathbb{R}$ with $|\mu(t)| \leq \frac{\gamma}{2} + D\Delta T$ and $|\rho(t)| \leq F\Delta T + \frac{\gamma}{2}$ for all $t \geq 0$. Equation 4 provides error bounds for approximating the values output by the digital accumulators with continuous integrators. For the digital PLL design, the integrand for the c-integrator is either $+1$ or -1, and its discretization is exact. Furthermore, the discretization for the values of c and v are accounted for by the ρ terms of their integrators. Hence, we can simplify Equation 4 and let $\mu = 0$ for the digital PLL model. SpaceEx supports linear differential inclusions, so, the ρ functions are easily added to the model for computing c and v. Once again, SpaceEx quickly establishes global convergence.

Non-linearities of the DCO. As described in Section III, the DCO is fundamentally non-linear in its response to its control inputs, c and v. For our model, we considered c in a range of 0.9 to 1.1 and v in a range of 0.1 to 2.5 (arbitrary units). The range of c matches what we could infer from [1]. The range

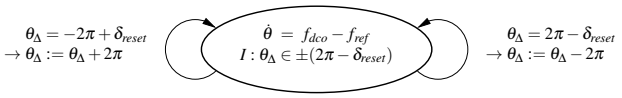


Fig. 6. A hybrid automaton for the linear PFD

of v is definitely wide; we chose it to show the flexibility of our approach. We divided v into seven overlapping intervals: when a trajectory leaves one region it arrives in the *center* of next interval – this prevents chattering mode-transitions that would cause SpaceEx to time out. For each interval, we computed a linear approximation for f_{dco} as a function of c and v, bounded the error, and incorporated the error terms into the linear differential inclusions for $\dot{c}$.

The bang-bang PFD. First, consider the linear PFD. It can detect phase differences of up to nearly a full clock period in either direction. For example, if the up signal goes high nearly a clock period before the dn signal, that indicates that the DCO is nearly 2π radians behind the reference. At the other extreme, the dn signal goes high nearly a clock period before the up signal to indicate that the DCO is nearly 2π radians ahead. Figure 6 shows a hybrid automaton model for the linear PFD.

Note that during the time that the reset signals are asserted (see Fig. 4), the PFD may fail to acquire an edge of one of the clocks.

A simple model of the digital PLL could be obtained by creating the product automaton from each of the component automata described above. To verify global convergence, it suffices to show that this product automaton converges to correct phase and frequency lock from all initial states. However, this results in a *huge* number of mode transitions. The key issue is that while the output of the PFD could have a mode transition for nearly every cycle, the value of v changes quite slowly. Thus, SpaceEx would need to analyse a large number of mode changes before v settles; in practice, this results in a time out. Furthermore, SpaceEx adds an error-term in all directions to the reachable space with each mode transition. Because v changes very little between mode transitions of the PFD, these error-terms overwhelm the convergence of v. To avoid these limitations, we created a model for the digital PLL that consisted of three sub-models according to how "far" the PLL is from its lock state.

Model 1: (the blue path segment in Fig. 5). This model is for the region where v is much lower than the equilibrium value. We construct a model that has c and θ_Δ as state variable and models v as a static interval. A complication to this argument is that due to delays in the reset loop of the PFD, the PFD may occasionally report that the DCO leads the reference, causing c to temporarily increase – these anomalous flows are depicted by the red arrows in Fig. 5. These anomalies are captured by our model, and SpaceEx shows that c moves to a small, invariant interval containing $c_{\min}$. Because $c < c_{center}$, $\dot{v} > 0$, from which we conclude that v increases, and the entry conditions for model 2 will eventually be satisfied. We use

an equivalent construction when v is much greater than the equilibrium value. Note that v must be analysed separately from c and θ_Δ to avoid the issues with different time scales for v and θ_Δ.

Model 2: (the magenta path segment in Fig. 5). For v in these bounds, the linear phase path bounds θ_Δ and ensures that there are no anomalies like those described for Model 1. Here we use the product-automata construction described earlier; c, v, and θ_Δ are all included as state variables. SpaceEx shows that v continues to progress towards its equilibrium value, and in so doing verifies our choice of bounds for v. In other words, our manual calculation was helpful to obtain a successful verification, but the soundness of the verification does not depend on the correctness of these calculations.

Model 3: (the green path segment in Fig. 5). Here, $\frac{1}{N} f_{dco} = f_{ref}$, and c and v follow a zig-zag path to settle at their equilibrium values. Along this path, θ_Δ frequently changes sign, causing a large number of mode transitions that would obscure the progress of v in the SpaceEx analysis. Our linearized model for the DCO frequency is

$$f_{dco} \quad \in \quad a_v v + a_c c \oplus Err \tag{5}$$

where $\oplus$ denotes Minkowski sum[1], and Err is an error-bound interval for the linear approximation.

$$w \quad = \quad \frac{a_v v + a_c c}{N} - f_{ref} \tag{6}$$

and construct a model whose state variables are w and θ_Δ. SpaceEx readily shows that w and θ_Δ both converge to intervals around 0.

Now, note that if $|w|$ is small, then given v, we can derive tight bounds for c. This allows us to construct a model, using a small interval for w, that shows that v (and therefore c) converges to its equilibrium value.

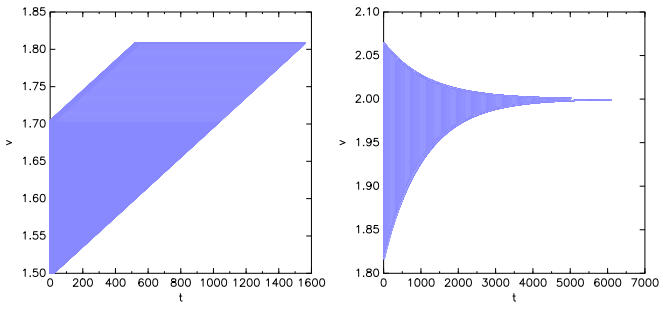
Together, these results from SpaceEx show that all trajectories that start in the valid region for model 1 eventually enter the valid region for model 2. All trajectories that start in the valid region for model 2, eventually enter the valid region for model 3. All trajectories that start in the valid region for model 3, converge to the desired equilibrium point. Because the union of the valid regions for models 1, 2, and 3 covers the entire state space for c, v, and θ_Δ, global convergence of the digital PLL is verified. As an example of the process, Figure 7 shows how v converges to its equilibrium value in models 2 and 3.

B. Limitations of the model

Our verification is relative to the model, and the model makes several simplifications relative to the real circuit. This section summarizes the most important of these simplifications.

[1] The Minkowski sum of two sets, A and B is the set of elements that can be obtained as the sum of an element from A and an element from B:

$$A \oplus B \quad = \quad \{z \mid \exists a \in A.\ \exists b \in B.\ z = a + b\}$$



v vs. t in Model 2 v vs. t in Model 3

Fig. 7. SpaceEx plots showing convergence of v for Models 2 and 3

Our model omits the Delta-Sigma modulator, the direct phase control path and the low-pass filter of the complete design. **Modeling the Delta-Sigma modulator** should be straightforward using the quantization model from Eq. 4.

The linear phase control path. We included a simple, linear model of the "time gain" of the linear phase control in our model. The phase shift of this path is applied once for every cycle of the PFD. This means that the phase shift is proportional to both the phase offset and the minimum of f_{ref} and $\frac{1}{N} f_{dco}$. Our linear model is valid when the PLL is close to lock, and we plan to model the non-linearity of this path in future work.

The low-pass filter seemed like an obvious candidate to include in our model for SpaceEx: we can model it as a purely linear system with three state variables and no mode transitions. When we included the filter in the PLL model, SpaceEx failed to show convergence. We suspect that the filter's low cut-off frequency results in a stiff model. We hope to explore this further and possibly along the lines of those presented in [12] to model the PLL with the low-pass filter.. An interesting opportunity here is that if the cut-off frequency of the low pass filter is close to that of the v-integrator, the PLL will be unstable. We intend to use this as a test case to show how our methods can identify a faulty design.

The PFD has a metastability issue that is hidden by the abstraction that we used. Basically, there are situations where the PFD must "decide" between reseting θ_Δ to 0 or continuing with a value that is close to $\pm 2\pi$. While the issue of metastability occurs in any PLL design, we have not seen it mentioned or addressed in the verification literature.

Finally, our model has **fixed coefficients** for the linear differential inclusions that model the PLL components. A real design will not exactly match any pre-specified value for these parameters, and they will be specified as intervals instead. SpaceEx only supports models where the coefficients are fixed, real numbers. In the next section, we introduce an approach that allows verifying global convergence under the more general realistic condition of having interval bounds for key model parameters.

V. Parameterized Verification with Z3

Consider the problem of showing that all trajectories in an invariant region Q_0 eventually reach a target region Q_T with $Q_T \subseteq Q_0$. This can be proven by exhibiting a *Lyapunov function*, Φ, that satisfies the two conditions below:

1) $\forall x \in Q_0 - Q_T.\ \Phi(x) > 0$; and
2) $\forall x \in Q_0 - Q_T.\ \frac{d}{dt}\Phi(x) < 0$.

Because Q_0 is invariant, all trajectories that start in Q_0 will remain in Q_0 forever. Furthermore, if the trajectory has not entered Q_T, $\Phi(x)$ strictly decreases with time along any trajectory outside of Q_T. Therefore, $\Phi(x)$ must eventually be less than or equal to zero. Because $\Phi(x)$ is strictly positive outside of Q_T, the trajectory must eventually enter Q_T.

The soundness of the Lyapunov argument does not depend on how Φ was obtained; it only requires that Φ satisfy the Lyapunov conditions. In this section, we borrow an approach from linear systems theory to construct a Lyapunov function, and use the Z3 SMT solver to show that the Lyapunov conditions stated above hold for this function when used with the non-linear model for the digital PLL. By using this approach with interval bounds for key model parameters, we show that the verification holds for *any* digital PLL whose components implement the model within the given bounds. We observed that a direct application of this method produced a system of non-linear relations where the SMT solver did not terminate in a practical amount of time. However, we can modify the original function to produce a new function that Z3 can show satisfies the Lyapunov conditions above. This verifies global convergence for any implementation of the PLL whose parameters are within the interval bounds.

As a preliminary experiment, we considered showing convergece from states where $\frac{1}{N}f_{dco}$ is much different than f_{ref}. In this case, the PFD acts like a frequency comparator, and we considered a simplification of Eq. 3 without θ and where $\dot{c} = g_1(f_{ref} - \frac{1}{N}f_{dco})$. Here, we use a non-linear DCO from Eq. 2.

To construct Φ, first consider a linear system, $\dot{x} = Ax$. Let P be the solution of

$$A^T P + PA \quad = \quad -I \tag{7}$$

By construction, P is symmetric. If P is positive definite, then the system $\dot{x} = Ax$ globally converges to $x = 0$ [13, p. 154]. To prove this, observe that $\Phi(x) = x^T Px$ satisfies the Lyapunov conditions.

Next, consider the PLL model with a non-linear DCO model and saturating accumulators as described in the previous section. Let h be the time-derivative function for this model, in other words, $\dot{x} = h(x)$ where $x = [c\ \ v]^T$. Let x_0 be the desired operating point of the PLL; in particular $h(x_0) = 0$. To show global convergence, let $A = Jac(h, x_0)$ be the Jacobian of h when evaluated at x_0; let P be defined as in Eq. 7; and let $\Phi(x) = x^T Px$. The PLL globally converges to x_0 if P is positive definite and for all initial points $x \in Q_0 - \{x_0\}$, $\frac{d}{dt}\Phi(x) < 0$. Positive-definiteness can be tested by adding a conjunct with that constraint to the solver and showing that a suitable P exists. The second part is equivalent to showing $\forall x \in Q_0 - \{x_0\}.\ h(x)^T Px < 0$. Z3 solves this problem in a few seconds, including the multiple cases in the definition of h to account for saturation of the accumulators.

We attempted to repeat this analysis using interval bounds for the parameters α, β, and f_0 for the DCO from Eq. 2, allowing each parameter to vary $\pm 20\%$ from its nominal value. With our initial attempt, Z3's solver failed to complete. While modern SMT solvers can handle non-linear relations, the computational cost grows extremely rapidly with the number of non-linear terms. Accordingly, we sought to simplify our Lyapunov function. Using the Jacobian based approach defined above:

$$A_0 \quad = \quad f_0 \begin{bmatrix} \frac{g_1 f_{ref}\beta}{1+\beta c_{center}} & -\frac{g_1 \alpha}{1+\beta c_{center}} \\ g_2 & 0 \end{bmatrix} \tag{8}$$

As described above, one can propose any matrix for A, and if function obtained by solving for P satisfies the Lyapunov conditions, global convergence is established. Thus, we looked for ways to "simplify" A_0 to obtain a system of inequalities that would show convergence and be tractable in Z3. Noting that the nominal values for α and β are both one, we factored them out from the numerators in the elements for the first row of A_0 to get

$$A_1 \quad = \quad f_0 \begin{bmatrix} \frac{g_1 f_{ref}}{1+\beta c_{center}} & -\frac{g_1}{1+\beta c_{center}} \\ g_2 & 0 \end{bmatrix} \tag{9}$$

With this change, Z3 verified the Lyapunov conditions, again in just a few seconds.

Now, consider adding error terms as described in Section IV to the derivatives to obtain an inclusion. These error terms perturbed the effective values of c and v in the calculation of the derivative. Let Err denote these error terms, and assume that Err is symmetric about 0: if $\eta \in Err$ then $-\eta \in Err$ as well. We now want to show:

$$\forall X \in Q_0 - Q_T.\ \forall \eta \in Err.\ h(x+\eta)^T Px < 0$$

From the symmetry of Err, this is equivalent to showing

$$\begin{array}{l} \forall X \in (Q_0 - Q_T) \oplus Err.\ \forall \eta \in Err. \\ \quad (x + \eta \in Q_0 - Q_T) \Rightarrow (h(x)^T P(x+\eta) < 0) \end{array} \tag{10}$$

The last form is easier for the SMT solver because it moves the η term out of the non-linear function, h. With the earlier models, Z3 showed convergence to the point x_0. With this model, Z3 shows convergence into a small rectangle that contains x_0. This rectangle is larger than $x_0 \oplus E$ because the disturbance can be time-varying.

Finally, we combined using interval bounds for the model parameters and including error terms in the derivative function. Again, Z3's solver failed to converge. This time we noted that the denominator of the elements in the first row of A_0, $1 + \beta c_{center}$ is always positive. Thus, we can multiply the second inequality of the Lyapunov conditions by $1 + \beta c_{center}$ to obtain the equivalent condition:

$$\begin{array}{l} \forall x \in (Q_0 - Q_T) \oplus Err.\ \forall \eta \in Err.\ (x + \eta \in Q_0 - Q_T) \\ \quad \Rightarrow ((1 + \beta c_{center}) h(x)^T P(x+\eta) < 0) \end{array} \tag{11}$$

Using this formulation, Z3 quickly verified global convergence with interval bounds for model parameters and error terms in the derivative function.

VI. Conclusions and Future Work

We have shown global convergence for a digital phase locked loop (PLL). We modeled the components of the PLL using piecewise linear differential inclusions, and then showed that all initial states converge to a small region near the intended operating point. These component models included non-linearities in the digitally controlled oscillator, saturation and quantization effects in the accumulators, and modeling of both a linear and a bang-bang phase-frequency detectors. Using a simplified model, we showed how the convergence results can be extended to the case where the specifications for components are given as interval bounds rather than exact values.

We chose to use SpaceEx [4] for the reachability computations because it was designed from the outset to handle large linear hybrid automaton models. The piecewise linear inclusions model the PLL components quite well. On the other hand, we encountered problems with long compute times and large over approximations when SpaceEx computed non-trivial fix points for cycles of modes. The solution we found was to organize the modes of the automaton according to the typical behaviour of the PLL during lock to avoid cycles of modes. With thise changes SpaceEx could verify global convergence in a few minutes.

SpaceEx requires fixed values for the model coefficients. We also showed the convergence can be established using Lyapunov functions, and the correctness of these functions can be shown with an SMT solver. For the work reported here, we used Z3 [5]. Here too, we encountered issues of time-outs: the solver would either complete in a second or two, or they would go on for hours without terminating. In this case, the solution was to manually simplify the function. This works particularly well with the Lyapunov approach; there's no way to introduce an error by simplifying a proposed Lyapunov function. If an inappropriate change is made, the proposed function will be refuted. Our SMT-based method is at a relatively preliminary stage and we are interested in seeing if we can apply it to a model that is as detailed as the one that we used with the hybrid-automata appproach.

There are many areas for future work. We would like to provide bounds on lock time (excluding metastability). Then we plan to complete models for the low-pass filter and the Delta-Sigma modulator. We plan to examine other digital PLL architectures to assess how much of the effort from this verification can be re-used for other designs. We expect that the re-use will be large, but we do not expect full automation given the need to guide the tools away from problems of time-outs and over approximations.

A very promising follow-on is to formalize the connection between the models used here and those used in other phases of the analog and mixed-signal design process. For example, we used "designer" provided models of the main components of the PLL. How do we know that these handwritten models correspond to the actual circuit? Thus, we want to connect this work with component validation.

Acknowledgments

We appreciate feedback from designers who have given us feedback on PLL design and verification, especially Elad Alon, Brian Casper, Frankie Liu, Frank O'Mahony, and Suwen Yang. This work has been supported by grants from Intel and from NSERC Canada.

References

[1] J. Crossley, E. Naviasky, and E. Alon, "An energy-efficient ring-oscillator digital PLL," in *Proceedings of the Custom Integrated Circuits Conference (CICC'2010)*, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1109/CICC.2010.5617417

[2] J. Kim, M. Jeeradit, B. Lim, and M. A. Horowitz, "Leveraging designer's intent: a path toward simpler analog CAD tools," in *Proceedings of the Custom Integrated Circuits Conference (CICC'2009)*, Sep. 2009, pp. 613–620. [Online]. Available: http://dx.doi.org/10.1109/CICC.2009.5280741

[3] K. S. Kundert, "Introduction to RF simulation and its application," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 9, pp. 1298–1319, 1999. [Online]. Available: http://dx.doi.org/10.1109/4.782091

[4] G. Frehse, C. L. Guernic *et al.*, "SpaceEx: Scalable verification of hybrid systems," in *Proceedings of the 23rd Conference on Computer Aided Verification*, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_30

[5] L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[6] H. Lin, P. Li, and C. J. Myers, "Verification of digitally-intensive analog circuits via kernel ridge regression and hybrid reachability analysis," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 66:1–66:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488814

[7] I.-S. Dhingra, "Formalising an integrated circuit design style in higher order logic," Ph.D. dissertation, Computer Laboratory, Cambridge University, Nov. 1988. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-151.html

[8] Z. J. Dong, M. H. Zaki, G. Al-Sammane, S. Tahar, and G. Bois, "Checking properties of PLL designs using run-time verification," in *International Conference on Microelectronics (ICM'2007)*, 2007, pp. 125–128. [Online]. Available: http://dx.doi.org/10.1109/ICM.2007.4497676

[9] Z. Wang, N. Abbasi, R. Narayanan, M. Zaki, G. Al-Sammane, and S. Tahar, "Verification of analog and mixed signal designs using online monitoring," in *Mixed-Signals, Sensors, and Systems Test Workshop, 2009. IMS3TW '09. IEEE 15th International*, 2009, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IMS3TW.2009.5158695

[10] A. Jesser and L. Hedrich, "A symbolic approach for mixed-signal model checking," in *Proceedings of the 2008 Asia and South Pacific design automation conference (ASPDAC'08)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 404–409. [Online]. Available: http://dl.acm.org/citation.cfm?id=1356802.1356903

[11] M. Althoff, A. Rajhans *et al.*, "Formal verification of phase-locked loops using reachability analysis and continuization," in *Proceedings of the 2011 International Conference on Computer Aided Design*, Nov. 2011, pp. 659–666.

[12] C. Yan, M. R. Greenstreet, and J. Eisinger, "Formal verification of an arbiter circuit," in *Proceedings of the 16th International Symposium on Asynchronous Circuits and Systems*, 2010, pp. 165–175. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2010.25

[13] P. J. Antsaklis and A. N. Michel, *A Linear Systems Primer*, 1st ed. Birkhauser Basel, 2007. [Online]. Available: http://dx.doi.org/10.1109/MCS.2009.932913

Formal Co-Validation of Low-Level Hardware/Software Interfaces

Alex Horn*, Michael Tautschnig*, Celina Val†, Lihao Liang*,
Tom Melham*, Jim Grundy‡, Daniel Kroening*
*University of Oxford †University of British Columbia ‡Intel Corporation

***Abstract*—Today's microelectronics industry is increasingly confronted with the challenge of developing and validating software that closely interacts with hardware. These interactions make it difficult to design and validate the hardware and software separately; instead, a verifiable co-design is required that takes them into account. This paper demonstrates a new approach to co-validation of hardware/software interfaces by formal, symbolic co-execution of an executable hardware model combined with the software that interacts with it. We illustrate and evaluate our technique on three realistic benchmarks in which software I/O is subject to hardware-specific protocol rules: a real-time clock, a temperature sensor on an I^2C bus, and an Ethernet MAC. We provide experimental results that show our approach is both feasible as a bug-finding technique and scales to handle a significant degree of concurrency in the combined hardware/software model.**

I. Introduction

A growing problem for today's microelectronics industry is co-design of hardware alongside embedded, low-level software that closely interacts with it. In particular, semiconductor designs are witnessing an increased use of on-chip microcontrollers running firmware to implement functionality that would formerly have been implemented in hardware. This trend is driven by factors that include the following:

- Extracting the control of complex devices and implementing it in firmware can cut development schedules while adding flexibility and survivability.
- By making on-chip devices more capable, work can be shifted away from the CPU, where performance is increasingly hard-won. The richer control required for more capable devices further drives the trend.

The sorts of devices that are typically integrated on-chip are controllers for power management, hardware with sequestered functionality for remote management or secure content, and increasingly capable graphics processors.

Co-design and validation of such devices together with their firmware, with the predictability needed to schedule fabrication and hit market windows, has become an acute challenge. Similar challenges arise in developing firmware for systems on chip (SoCs) and general embedded systems [1].

Firmware is just hardware-specific software. One might therefore expect that the problem can be addressed by some combination of today's separate techniques for design and verification of hardware and software. But the results of this approach are disappointing. The problem is the complex nature of the interactions at the hardware/software interfaces. All large systems are structured into subsystems, but the interfaces between hardware and software subsystems are more problematic than those in a homogeneous system.

- In a homogeneous design, the documentation of interfaces (say by header files) is both understandable by developers and processable by tools. A compiler, for example, can guarantee some consistency in how two modules view a shared interface. But hardware and software are typically described in different languages, processed by separate tool chains. And the hardware and software design teams have their own descriptions of the interfaces they share, with little to ensure the two are consistent.
- The mechanisms for invoking functionality and sharing data among modules in a homogeneous system, particularly in software, are relatively few. In contrast, the means of passing information between hardware and software are varied and built up from nonstandard primitives that may include interrupts, memory-mapped I/O, and special-purpose registers. The situation is analogous to software before procedure-calling conventions were standardized.
- The hardware/software interface also marks a boundary between different threads of concurrent execution. Without the shared understanding of synchronization that follows from a common language and library, concurrency at the hardware/software interface needs special treatment.
- Finally, a hardware/software interface almost always marks a boundary between different teams, working in different parts of a company and having different educational backgrounds and skills. The scope for misunderstanding is greater than usual.

The challenges faced by those implementing the two sides of a hardware/software interface are high—but so is the need to get it right. Building a system with a new interface and then testing it to find and remove bugs is a perilous practice. When designers move the control from a complex hardware device into firmware, the stripped-down hardware can be difficult to test without a means to run the firmware. Testing the hardware and firmware together is difficult before silicon is fabricated: simulators are slow, emulators expensive, and FPGAs limited in capacity. Delaying extensive testing until silicon is available is unacceptable as it serializes the development of hardware and software to the point where it can be difficult to meet the

Supported by ERC project 280053 and EPSRC project EP/H017585/1.

ISBN 978-0-9835678-3-7/13.

project schedule. And, of course, any hardware bugs found at this stage will be expensive to fix.

An increasingly common approach to designing embedded software is to employ *virtual prototyping*, so that coding can begin before silicon is available. A software model (in C or SystemC) serves as a proxy for the hardware on which the embedded firmware code can be developed and tested. In our research, we leverage this trend and propose a new approach to co-validation of hardware/software interfaces by *formal, symbolic co-execution* of two combined pieces of code: a software model of the hardware, and the real embedded firmware that interacts with this hardware. When necessary, we capture the parallelism between firmware and hardware with a modelling approach that employs software concurrency in the form of asynchronous threads.

We demonstrate our idea with three realistic benchmarks, all publicly available on the web.[1] These are constructed by combining hardware models adapted from a virtual machine emulator and software taken from a general-purpose OS kernel. The interactions at the hardware/software interfaces of these benchmarks are characteristic of devices of interest to our industry partner—including low-level software, akin to a driver, executing on a separate on-chip microcontroller. Our experimental results show the approach both is feasible as a bug-finding technique and scales to handle a significant degree of hardware/software concurrency.

A further, methodological, contribution of our work is the identification of the open-source QEMU [2] code base, together with Linux device drivers [3], as a rich source of characteristic examples to drive research in this domain. The correctness properties we have devised also exemplify validation problems that are typical of low-level hardware/software interfaces. Co-validation poses distinctive challenges and has growing industrial importance; it is our hope that this work will encourage other formal verification researchers to engage with this important problem in contemporary hardware design.

II. Validation Aims and Technical Approach

We have designed and experimentally evaluated a method for semi-automatically searching for bugs in the interactions at hardware/software interfaces. Our aim is to find potential violations of certain correctness properties as the hardware interacts with the low-level software under scrutiny. Such violations yield counterexample traces that can expose bugs both in the low-level software and in the hardware model. Our method targets early and relatively high-level *software* models of hardware, which need not be cycle accurate.

For our work, the purpose of hardware models is to capture the hardware side of interactions that occur at nonstandard hardware/software interfaces. This includes modelling complex, ah-hoc side-effects characteristic of low-level devices. For example, when the software reads a hardware register, this may cause other changes to the visible state at the interface.

[1] http://www.cprover.org/firmware

In general, these interactions are expected to conform to a protocol that can be articulated in terms of pre- and post-conditions, which we refer to as *properties*. We formalize these as runtime assertions within the executable hardware model itself. Specifications of interface protocols are usually articulated separately, in logic. Our approach, however, is designed to appeal to practicing engineers who use virtual platforms to test their intuition through co-simulation, and so we embed specifications within executable models—which are given formal meaning through a bit-precise execution semantics [4].

A. Modelling Approach and Concurrency

In the conventional approach to formal hardware verification, hardware models are essentially state-transition systems: a formal model is given that determines or constrains the next state of registers/memory in relation to the current state and inputs. In this work, we propose a different approach that need not be cycle accurate and provides a higher-level, event-driven software abstraction of the hardware, focussed on interactions at its interface with low-level software.

In some hardware/software systems, the speed of the hardware and low latency of the interaction mechanism, relative to software execution, imply that we do not need to model hardware/software concurrency. An example is the RTC benchmark in Section IV. In these cases, our combined code, comprising the firmware plus the hardware model, can simply represent interaction by procedure calls into the hardware model from the software. This simplification encodes an assumption that hardware response is effectively instantaneous.

More interesting is our handling of hardware/software concurrency in the cases where it matters. Here, we model the hardware by asynchronous software threads invoked by the low-level code that interacts with the hardware. When this code engages in an interaction with the hardware—writing to a register, say—a concurrent, asynchronous thread is spawned whose sole purpose is to call a function in the hardware model that initiates an execution to model the hardware's response. Once the function returns, the thread terminates. This event-driven abstraction enables us to find concurrency bugs, as illustrated by the Ethernet MAC benchmark in Section VI. It also maps well onto the early-stage hardware modelling activity that our validation method targets.

In the specific context of our benchmarks drawn from QEMU, we are able justify making threads of the hardware model atomic with respect to each other and the low-level software. This reduces the complexity of our analysis. We recognize, however, this will not always be possible.

The low-level software's response to hardware *interrupts* are also modelled by asynchronous threads, this time created at run-time by the hardware model—and not atomic. This allows us to capture concurrency bugs in interrupt handling, a prominent potential source of errors in the industrial systems we have in mind.

B. Validation Technology and Concurrency Encoding

Our validation method is designed to leverage today's highly optimized SMT/SAT solvers. Where our combined code is purely sequential, we can analyse it with any software analysis tool that gives a bit-precise semantics to C and can check our embedded run-time assertions. In Section VII, we report experimental results using CBMC and, for comparison, also using KLEE—a pathwise testing tool that achieves high coverage through symbolic execution [5].

Our main approach, however, is to analyse our coalesced code by symbolic execution with aggressive path-merging, as exemplified by CBMC [6]. This yields a mathematical formula that encodes multiple execution paths up to a certain depth, which is then checked by a SAT/SMT solver. This approach maximizes the exploitation of today's optimised solvers and avoids the potentially exponential number of execution paths explored by path-wise enumeration [7], [5].

To handle concurrency, we exploit a recent encoding of concurrent software execution in CBMC that uses *partial orders* to constrain the relative timing of events that access shared state [8]. Roughly speaking, it works as follows. Accesses to shared state by separate concurrent threads are first decoupled by being given distinct symbolic references. An integer *clock* is introduced for each access to shared state, and a partial order is given among the clocks that encompasses all feasible interleavings of these accesses to shared state. Finally, order-dependent equality relationships are established among the values named by the decoupled references, making connections between the state values 'seen' by the hardware model and low-level software. All this is efficiently encoded in a quantifier-free formula whose size is cubic in the maximum number of shared state accesses. Any satisfying assignment found by a SAT/SMT solver corresponds to a property violation.

Encoding concurrency by partial orders side-steps having to deal explicitly with the complexity of interleavings, and produces a highly competitive analysis for concurrent software [8], which we exploit in this work. It also allows the ordering of accesses to be more loosely specified than in conventional sequential consistency. The latter property is used in [8] to capture the complex semantics of weak memory models in modern multi-core architectures, but is not (yet) exploited in our work on hardware/software co-validation.

III. Hardware Models from QEMU Devices

To evaluate our method, we extract hardware models from the open-source QEMU virtual machine emulator [2] and combine them with Linux device drivers [3]. Our idea is to leverage the rich collection of hardware models that QEMU provides, in combination with real OS driver code. This strengthens the objectivity and realism of our experiments, since they retain essential characteristics of production code. This includes a specific division into hardware models and low-level software, which moreover originate from separate developer communities.

QEMU was designed for hardware virtualization, not experiments in formal co-validation, and extracting usable stand-alone hardware models from QEMU code is not entirely straightforward. To give an idea of what is involved, we briefly sketch some aspects of the QEMU architecture, before going on to present our benchmark experiments.

QEMU is written in C. Each QEMU virtual machine is divided into *boards*, each of which consists of *device* and *bus* models. Communication between device models can occur only through bus models. This is the basis for a modular design, implemented through a QEMU-specific factory and service locator pattern known as QDev [9].

QDev organizes hardware models into a dynamic tree data structure that relies on a QEMU-specific object model called QOM [10]. In essence, QOM seeks to extend C with object-oriented programming features. To achieve this, QOM stores information about its internal C structures in `glib` trees and hash tables. An instantiation of such a structure is called an *object*. Function pointers serve as methods. QEMU's physical memory management architecture determines which object methods of a device model are called when memory regions are accessed by the guest operating system [11].

For our benchmarks, we extracted stand-alone C hardware models by excluding all physical memory management code and dependencies on QDev and QOM. This was done through a somewhat laborious process of careful slicing and approximation of essential features.

By default, QEMU accelerates dynamic code translation though just-in-time compilation [12]. For our purposes, this can fortunately be bypassed through an undocumented feature called QTest, a client-server architecture that facilitates testing of hardware models. Few QEMU models currently take advantage of this test harness, but the trend is towards more testing. The benefit for our approach is that these tests can give insight into hardware-specific verification properties, and serve as starting points for symbolic co-execution.

IV. Co-validation of a Real-Time Clock

Our first benchmark is the MC146818 real-time clock (RTC), a low-power CMOS device that provides—among other functionality—a time-of-day clock, a calendar, and programmable timers for periodic interrupts and square-wave generation [13]. One of the stated purposes of the MC146818, which is quite an old device, is to 'relieve the [micro-processor] software of the timekeeping workload'. This motivation also drives today's proliferation of complex on-chip device controllers, themselves running firmware.

The RTC is representative of hardware devices that firmware interacts with through special-purpose registers, a common low-level software/hardware interface idiom [14]. The speed of the RTC device means we can represent interaction with it by procedure calls in the firmware; in essence we can assume, in this benchmark, that both a register write and the hardware's response to it are instantaneous.

In this first benchmark, we focus on only part of the RTC interface: reading and writing the registers that hold the time, date, and alarm data. As will be seen, this is not simply a matter of the firmware executing a 'read' or 'write' instruction, but requires some ancillary manipulation of bits in control

status registers. Our validation task is to check for violation of the protocols that govern this mechanism.

A. Interface Properties

The MC146818 presents its interface as 64 bytes of RAM, addressed `0x00` though `0x3F`. The time, date, and alarm data are held in the first 10 bytes, each of which is essentially a 'register' at a specific address [13]. For example, the byte at `0x09` is the register that holds the year. To access a register, software must execute a sequence of two I/O instructions that access two different memory-mapped hardware registers at addresses `0x70` and `0x71`. The first determines the data register to be accessed, and the second holds the value of this register. It is an error to read or write a data register value at address `0x71` without first setting the register to be accessed:

- ⋆ *Each execution of* `outb 0x71` *or* `inb 0x71` *must be preceded by a unique* `outb 0x70`.

This property alone is insufficient to guarantee safe writes of data. The firmware controlling this device must also correctly manipulate two control bits in 'Register B', one of four other RAM locations whose individual bits monitor and control a diverse assortment of device operations. The two relevant bits are the *SET* bit and the *data mode (DM)* bit.

- ⋆ *The SET bit of Register B must be enabled (have value* 1*) when any of the time, date, or alarm registers are written.*

Once SET is enabled, data can be safely written as either binary or binary-coded decimal. The choice must be made explicit by writing 1 or 0 to the DM bit. In addition, the selected 'data mode cannot be changed without re-initializing the 10 data bytes' [13]. A permissive interpretation of this sentence in the data sheet yields the following two properties:

- ⋆ *The DM bit can be changed only when the SET bit is already enabled, or as the SET bit is also being enabled or disabled when Register B is written.*
- ⋆ *If the DM bit has changed since the SET bit has been or is being enabled, then every time, calendar and alarm register must have been written at the moment when the SET bit is disabled.*

The final property we discuss here says there are no concurrent hardware writes to any of the time, date, or alarm registers while the SET bit is 1. Note that no such guarantee exists once the SET bit is disabled.

- ⋆ *While the SET bit is 1, when data d is written to a time, calendar or alarm register R, a subsequent read of R returns d.*

We have shown only informal statements of our properties to make the exposition accessible. In our actual method, we encode properties as runtime assertions in the RTC hardware model. Several other properties, omitted here for brevity, are included in our experiments. This yields an executable specification, through which we expose a real bug (Section VII).

B. Technical Details of the RTC Benchmark

To illustrate the architecture shared by all our benchmarks and explain how hardware models are extracted from QEMU, we give here some technical details for the RTC benchmark.

The full RTC model in QEMU depends on a large amount of code irrelevant to our properties, and is too complex to analyse formally. We therefore manually removed these dependencies, including the dependency on the i440fx PCI host. We also manually sliced away some code not representing actual hardware, such as QEMU timers. These simplifications, which we would expect in future to mechanize, preserve the core of the RTC model.

There are two loops in the RTC hardware model that are hard for CMBC to handle, but the functions that contain them are for RTC timer functionality that has nothing to do with our interface properties. We could therefore safely remove these function calls without affecting the validation results.

For the low-level software side, we used the dependency tracking capabilities of CBMC to pull together sufficient Linux driver code to exercise the hardware model. This was done in a semi-automated way that produces a coalesced C program containing the model together with a superset of the exact Linux code needed to drive the hardware features covered. The coalesced program has around 49k lines of C code.

Symbolic execution of the coalesced program has to proceed from a main function that actually invokes the driver. For this, we develop scenarios that invoke the driver in various ways. These were specially written for the RTC benchmark, but a merit of our approach in general is that we might instead leverage test cases created by developers, as long as these initialize the hardware model. Our code initializes the RTC with a non-deterministic value representing the time. After initialization, we call the Linux device driver function `get_rtc_time()` to read the time from the RTC. Finally, we call `set_rtc_time()` to write back the time just read. These calls induce state transitions in the hardware model.

A similar approach can be taken to produce benchmarks for stand-alone QEMU hardware models, in isolation from their driver, simply by wrapping each QEMU model with some C code that enacts driver I/O scenarios. In the RTC benchmark, each such scenario includes an initialization step that sets the time in the RTC to a non-deterministic value, represented in binary-coded decimal and constrained to be in the range given in the MC146818 datasheet [13]. For a sanity check of the properties, we created a buggy test case for the stand-alone hardware model that calls `inb(0x71)` before `outb(0x70,*)`, where $*$ is a non-deterministic value.

V. Co-validation of an I²C Temperature Sensor

The second benchmark features a temperature sensor [15], called 'TMP105', that is controlled by software through the I²C bus [16]. This allows us to experiment with properties that go beyond fixed-sized register updates. Our hardware model for the I²C benchmark incorporates the essential interaction constraints for these updates, so the benchmark does not need to include a model of the I²C bus controller.

The temperature sensor has four registers: an 8-bit configuration register, a 16-bit temperature register, and 16-bit lower and upper temperature threshold registers for hysteresis. Reading and, when applicable, writing of these registers is done over the I^2C serial bus. The registers have different sizes, so the number of transmitted bytes varies.

Individual bits in registers must conform to rules similar to those of the RTC. We show a few illustrative properties, again stated informally here but in practice encoded as run-time assertions in the hardware model.

The sensor can be shut down by writing a 1 to the least significant bit of the configuration register. This turns off continuous temperature measurements to save power. While the sensor is in this 'shutdown mode' individual readings can still be triggered by writing a 1 to the most significant bit of the configuration register. This leads to the following property:

⋆ *Each read of the temperature register is preceded by a write of a 1 to the most significant bit of the configuration register if and only if the temperature sensor is in shutdown mode.*

Writing 1 to the most significant bit of the configuration register merely triggers an individual temperature measurement; the bit itself is immutable and not affected by the write.

⋆ *When the most significant bit of the configuration register is read, it is zero regardless of any previous writes to it.*

The next property concerns the configuration register.

⋆ *After writing byte c to the configuration register, the next read gives a byte c' where $c'[i] = c[i]$ for $0 \leq i < 7$.*

That is, all bits of the old and new configuration value are pairwise equal, except perhaps the most significant bit.

Altogether, the bus and register properties amount to around two dozen runtime assertions. It was straightforward to encode these in the TMP105 hardware model extracted from QEMU: the TMP105 internal state is stored in a C structure that has fields to that implement its registers and store control information related to communication through the I^2C protocol.

VI. Co-validation of an Ethernet MAC

Our final benchmark concerns interrupt-driven software for an Ethernet MAC with a direct-memory access (DMA) ring [17]. We concentrate on the functionality of receiving Ethernet frames. Each incoming frame is called an *RX frame*. The Ethernet MAC can be configured to generate a hardware interrupt for each RX frame. We call this 'interrupt mode'. When interrupts are disabled but RX frames should still be processed, the software polls for incoming data.

Hardware/software concurrency is therefore important to model in the Ethernet MAC benchmark, because multiple frames can arrive simultaneously and the software reacts to interrupts generated by the hardware. These are handled using the modelling approach discussed in Section II, and produce a significant degree of concurrency in the coalesced model.

A noteworthy complication is that the software switches between interrupts and polling to improve performance [18]. Similar techniques are used for block devices with high data throughput, such as solid state drives. Switching between polling and interrupt mode is known to be error-prone, so this benchmark is a good exemplar for concurrency bugs due to interrupts in a producer-consumer scenario. This section explains one such bug and how the developers fixed it.

The OpenCores Ethernet MAC features 128 DMA buffer descriptors [17], each of which determines the memory that holds an Ethernet frame. Our benchmark code elides the details of DMA address translation; instead, we focus on how the software and hardware synchronize their updates to the DMA buffer. In the case of RX frames, the software sets bit 15 in a buffer descriptor to 1 when the associated DMA buffer can be overwritten by the hardware. Such a buffer descriptor is said to be 'empty'. The hardware clears bit 15 to signal to the software that the DMA buffer associated with a buffer descriptor contains a new RX frame. Despite its simplicity, this communication protocol is error-prone when interrupts are being re-enabled, as illustrated next.

Suppose there is at least one empty RX buffer descriptor. The software switches from polling to interrupt mode as soon as it detects no new RX frames. To do this, it reads bit 15 of the next available RX buffer descriptor. Suppose the current buffer descriptor is empty and so this bit is still 1. In this case, the version of the software with the bug continues by clearing all RX interrupt sources before re-enabling all RX interrupts.

Unfortunately, this algorithm can result in RX frames being delayed or even dropped. Figure 1 shows an example, in which an RX frame arrives just after the check for new RX frames but before the RX interrupt sources are cleared. This RX frame will not trigger an interrupt until another one arrives. In fact, if there are no other ones, the delayed RX frame is not even promoted to the socket layer. This happens when the driver is stopped, for example due to standby.

The following properties will expose this concurrency bug:

⋆ *When the software enables the MAC receiver, there exists at least one empty RX buffer descriptor.*

⋆ *The software must eventually process every RX frame. At the very latest, when it is stopped, all RX frames must have been processed.*

The crux of these properties is that the driver must detect any potentially lost frames. Figure 2 shows how the developer for the 'ethoc' driver in the Linux 2.6.38 kernel release fixed the bug, ensuring these properties are then satisfied.

These and several other properties were analysed using CBMC and the partial order encoding for concurrency. The scenarios we wrote to exercise these properties asynchronously invoke the hardware model to trigger new RX frames or force the MAC to become busy.

A few simplifications were made to enable analysis within reasonable time and memory bounds. Because the solver has no array logic built in, we had to reduce the maximum number of DMA buffers to eight and shrink their sizes to at most two bytes. For the same reason, the number of buffer descriptors in the hardware model was reduced to eight. Finally, we

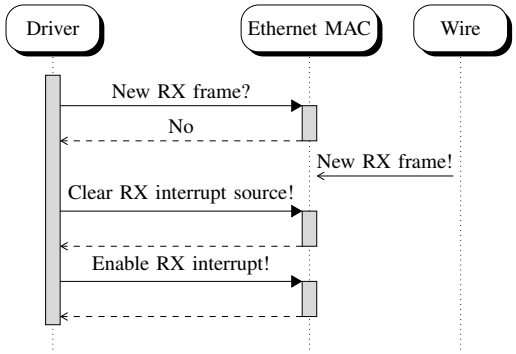


Fig. 1. Incorrect handling of an empty RX buffer descriptor causes potential package loss.

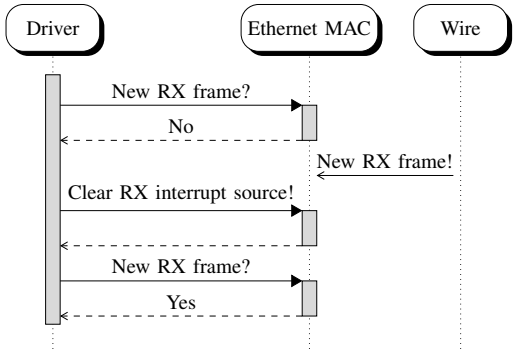


Fig. 2. A second buffer descriptor check, after the RX interrupt sources have been cleared, detects intermittent RX frame arrivals.

	LOC	#Unroll	#Threads	#Constraints	#Clauses	Sec.
RTC.1	47609	21	1	61314	25,797,536	122.5
RTC.1 (unfixed)	47609	21	1	61065	25,771,226	225.3
RTC.2	47609	21	1	61314	26,430,338	71.7
RTC.3	47609	21	1	60648	25,769,903	68.5
RTC.4	47609	21	1	60766	26,422,852	69.7
RTC.5	47609	21	1	60435	26,425,148	69.5
RTC.6	47609	21	1	60295	8,208,764	54.1
RTC.7	47609	21	1	60394	8,759,757	55.2
RTC.8	47609	21	1	60294	24,491,011	69.1
RTC.9	47609	21	1	60294	24,468,704	67.7
RTC.10	47609	21	1	60294	24,781,142	68.6
RTC.11	47609	21	1	60668	25,231,840	112.8
I2C.1	46609	16	1	159481	20,020,885	803.9
I2C.2	46609	16	1	158391	19,997,082	793.1
I2C.3	46609	16	1	158556	20,006,113	795.8
I2C.4	46609	16	1	158556	20,023,452	787.1
I2C.5	46609	16	1	158556	20,024,494	786.2
I2C.6	46609	16	1	158436	19,998,982	783.4
I2C.7	46609	16	1	158436	20,007,984	786.1
I2C.8	46609	16	1	158436	20,001,601	780.5
I2C.9	46609	16	1	163866	20,118,277	854.4
I2C.10	46609	16	1	164547	20,074,751	841.0
I2C.11	46609	16	1	162381	20,388,706	808.6
I2C.12	46609	16	1	158556	20,160,928	789.4
I2C.13	46609	16	1	158811	20,009,910	804.3
I2C.14	46609	16	1	160596	20,294,436	798.7
I2C.15	46609	16	1	160596	20,295,406	800.8
I2C.16	46609	16	1	158391	19,997,740	788.9
I2C.17	46609	16	1	167481	20,064,678	912.9
ETHOC.1+2	940	2	2	1036+57	336,557	1.7
ETHOC.1+3	940	2	13	4633+1063	8,109,581	46.1
ETHOC.1+4	940	2	17	6073+1300	17,192,339	145.7
ETHOC.5	2097	1	19	16707+20991	250,371,908	1680.7
ETHOC.6	2097	1	19	16683+21034	252,154,414	335.5
ETHOC.7	2097	1	19	16635+20750	239,259,859	219.5
ETHOC.5-seq	2097	1	1	17710	73,388,552	426.8
ETHOC.6-seq	2097	1	1	17686	73,324,230	426.1
ETHOC.7-seq	2097	1	1	17648	71,998,722	435.3

TABLE I
EXPERIMENTAL RESULTS

suppressed interrupts on changes of the interrupt mask because this functionality appears to be QEMU-specific and not part of the Ethernet MAC itself.

VII. EXPERIMENTAL RESULTS

Table I summarizes our experimental results. We performed all experiments on a 64-bit machine running Linux 3.5.0 with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory.

For the RTC and I^2C benchmarks, we employed sequential, multi-path symbolic execution. Loops were unrolled a bounded number of times. This type of symbolic execution generates a Boolean formula that encodes the expected interface properties and all calls to read and write procedures in the hardware model invoked from the low-level software. The formula is then checked with MiniSat 2.2. If the formula is satisfiable, a violation of some property has been found. Otherwise, no decisive conclusion about the validity of the property can be reached.

As part of our RTC experiments, we found a real bug in the QEMU hardware model[2] that causes it to violate property RTC.1 in Table I. The violation is exposed through a test that first writes a time or calendar register and then writes to one of the control registers of the device. For the combined RTC benchmark with the hardware model and driver code, CBMC reports the violation of property RTC.1 in $225.3\,s$, of which $177.4\,s$ were spent in MiniSat. The violation of property RTC.1 in the standalone RTC hardware model is found in $50\,s$.

The temperature sensor benchmark also helped to expose a real bug in the QEMU hardware model.[3] The bug causes data on the I^2C bus to be lost because of an off-by-one error. In the standalone I^2C hardware model, CBMC found a violation of property I2C.10 in $3.6\,s$ and property I2C.18 in $3.4\,s$, of which $1.4\,s$ and $1.3\,s$, respectively, were spent in MiniSat.

We also tried to use CBMC to expose property violations in the temperature sensor by analysing the combined driver code and hardware model. The scenarios developed for this analysis take a brute-force approach, in which we explore all possible sequences, of a fixed length, of non-deterministically chosen driver API function calls. The sequences are encoded symbolically, in a single run of the analysis. The idea was to simulate all possible fixed-length sequences of invocations of the driver API, such as might arise in a typical interrupt-driven setting. Our experiments with a bound of 15 driver API calls in the test harness failed to expose a property violation. CBMC timed out after $1800\,s$ when the number of API function calls was set to 20.

For the ETHOC device benchmark, our tool processes the coalesced code and symbolically encodes concurrent memory accesses as partial orderings, as discussed in Section II. The resulting formula, sent to an SMT/SAT solver, captures all possible execution schedules for the threads that enact

[2]http://git.qemu.org/?p=qemu.git;a=commit;h=02c6ccc6dde90dcbf5975b1c

[3]http://git.qemu.org/?p=qemu.git;a=commit;h=cb5ef3fa1871522a08866270

invocations of hardware functionality. Interrupts generated by the hardware model also introduce concurrency that is included in the encoding, because the software's interrupt handler executes asynchronously from the hardware's point of view.

This modelling approach generates a significant degree of concurrency. Our scenarios result in up to 18 threads being spawned, yet we can validate most of the concurrent scenarios, explained in further detail below, in under 6 minutes (one scenario requires up to 29 minutes). The partial-order based encoding of [8] thus appears to be well suited for efficiently checking our models of combined hardware/software systems.

First, we validated the QEMU hardware model in isolation. In under 1 minute, despite 12 threads being spawned, we showed that our hardware model correctly simulates interrupts to be raised asynchronously. As our symbolic execution is limited to bug-finding, we had added a runtime assertion to state the converse; the counterexample we found constitutes evidence that our interrupt properties are not vacuously true.

With a more than tenfold increase in the number of clauses when analysing the combined hardware/software system, the burden on the underlying decision procedure rises significantly. We therefore experimented with both a purely sequential composition of the systems, corresponding to an assumption of instantaneous hardware operation, and a concurrent version, capturing all behaviour. In just over 7 minutes, our experiments confirm that the sequential version does not exhibit the erroneous behaviour described in Section VI. This failure to detect the bug illustrates how developing software without a realistic hardware model poses risks. The complete, concurrent system spawns 18 threads to simulate all interactions of the software with asynchronous hardware and interrupts. In 29 minutes we were now able to detect a property violation that exposes the presence of the bug in the combined hardware/software system.

Other Experiments using KLEE. We have also analysed all three benchmarks using KLEE [5]. On the RTC benchmark with the combined hardware model and driver software, KLEE times out after $1800\,s$. It also times out on the standalone, corrected RTC hardware model, but it can find the bug in the original model in $1\,s$. The temperature sensor driver code cannot be compiled into LLVM IR, but KLEE can check the corrected hardware model in $1\,s$. When we analyse the original temperature sensor hardware model, KLEE exposes the bug in $1\,s$. In addition, we confirmed that KLEE cannot detect the concurrency bug in the Ethernet MAC driver; with both the corrected and buggy version of the driver, it explores 25 paths in less than $30\,s$ and passes all seven ETHOC properties (whether violated or not).

VIII. Discussion and Future Work

In this section, we indicate some of the limits of our work and discuss some directions for further research.

The hardware models in all our benchmarks serve as an executable specification of the hardware/software interface. This is helpful to engineers, who test their intuition through simulation. An executable hardware model also gives flexibility in expressing properties. But, under symbolic analysis, this can also give rise to a diverse range of logical formulas for checking—making it hard to find an appropriate decision procedure. The problem might be mitigated by adopting an embedded contract language [19].

We found executable hardware models to be essential in discovering key properties of the hardware and software, and to exploring the interfaces between them. Our models are event-driven and phrased at a high level—executable software abstractions that strike a balance between modelling accuracy and tractability. An obvious goal is to verify them formally against a lower-level reference model. Our Ethernet MAC benchmark could be used as a research case study for this, in which executions of threads in the stand-alone QEMU model are compared with the OpenCores RTL Verilog model [17].

Our Ethernet MAC benchmark exemplifies the subtleties of interrupts by explaining a concurrency bug in a driver. For the automatic analysis, we relied on symbolic encoding of interactions between concurrently running model elements. But we have not yet considered nested interrupts, which would require extensions to both our models and the concurrency encoding. In future, we would also like to address interrupt priorities, such as FIQ interrupts on ARM. Our experiments also expose opportunities for automatic slicing algorithms that are aware of concurrency semantics.

Multi-threading in the Ethernet MAC benchmark frames some of our current research on improvements to the partial order concurrency encoding. Our experience suggests there is potential to delegate some of the generation of constraints that constitute the partial order on state accesses to the SMT solver itself, where it could be done incrementally.

IX. Related Work

Most research on verification of low-level software has focused on operating systems and drivers, with some prominent successes [20], [5], [21], [6], [7], [22]. There has also been some work on formal analysis of assembly code [23] and even binary drivers have been analysed [24]. There is, of course, a large body of literature on design and verification of embedded systems at a higher level [25]. In our work, we address *formal co-validation* of complex interactions between low-level software and on-chip hardware, using a novel technique that combines symbolic execution and partial-order encodings [8], bypassing the scalability limitations of partial-order reductions in earlier work (e.g. [26]). Our work represents the first demonstration of this approach to this important problem in contemporary systems design.

There has been some research, with aims similar to ours, using bounded model checking [27] and interval property checking [28]. The emphasis of both these efforts is on machine instructions and cycle-accurate hardware models, while ours aims at early validation before a cycle-accurate model is available. This is reflected in the fact that the work of [28] targets Verilog code, while ours revolves around higher-level models in C. Earlier work also analysed a more non-

deterministic C model through abstract interpretation [29], but with less sophisticated support for concurrency. More recently, an automata-theoretic co-verification technique has been applied to PCI drivers [30].

Other related work has used symbolic simulation and SMT to check equivalence between a software reference model and a system containing (restricted) C code that invokes data computations on reconfigurable streaming hardware modelled in Java [31]. Hardware/software concurrency is not represented; interaction is modelled by synchronous calls from the software into an API that loads and runs the streaming hardware designs. The aim is to establish correctness of the dataflow computations in hardware/software co-designs. By contrast, our work and modelling approach seek to uncover bugs in hardware/software systems that interact through concurrent, imperative modification of shared state.

X. Concluding Remarks

We have described a new approach to co-validation of hardware and low-level software, based on formal co-verification of an executable hardware model together with the software that interacts with it. We articulate key correctness properties that we expect interactions at the hardware/software interface to exhibit, and check these by symbolic execution. As our experiments show, the approach can be adapted to a range of interaction mechanisms—and it can expose bugs.

Systematic experimental research into formal co-verification of hardware and low-level software is hindered by the lack of realistic benchmarks accessible to academic researchers. Our work suggests a way to close this gap. We exploit the availability of well-developed open source virtual machine emulator code, from which one can extract a wide range of typical hardware models. These models can be made to work with Linux drivers, which serve as a proxy for typical firmware code. A practical benefit is that this facilitates collaboration with the systems community, whose insights helped us understand the problem space and expose real bugs. We suggest that a community effort to develop a benchmark suite, following our approach, would produce an invaluable resource to drive further academic research into firmware validation.

XI. Acknowledgements

This work is funded by a donation from Intel Corporation for research on *Effective Validation of Firmware*. We are grateful for illuminating discussions with our project partners: Alan Hu (UBC), Luke Ong (Oxford), Moshe Vardi (Rice), and Sharad Malik (Princeton). We thank Anthony Liguori (IBM), Paolo Bonzini (Redhat), and Andreas Färber (SUSE) for their comments on the QEMU mailing list.

References

[1] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, pp. 1411–1430, May 2012.

[2] A. Liguori, "QEMU emulator user documentation," http://wiki.qemu.org/download/qemu-doc.html, Jan. 2010.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, 2005.

[4] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang, "An efficient finite-domain constraint solver for circuits," in *Design Automation Conference (DAC)*. ACM, 2004, pp. 212–217.

[5] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Assoc., 2008, pp. 209–224.

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*. Springer, 2004, vol. 2988, pp. 168–176.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *SIGPLAN Notices*, vol. 40, pp. 213–223, June 2005.

[8] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient Bounded Model Checking of concurrent software," in *Computer-Aided Verification (CAV)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 141–157.

[9] P. Bonzini, "QEMU developer mailing list – qdev for programmers," http://lists.nongnu.org/archive/html/qemu-devel/2011-07/msg00842.html, Jul. 2011.

[10] A. Liguori, "QEMU wiki – QOM," http://wiki.qemu.org/Features/QOM, Jul. 2011.

[11] A. Kivity, "QEMU developer documentation on memory API," http://git.qemu.org/?p=qemu.git;a=blob;f=docs/memory.txt, Jul. 2011.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *ATEC*. USENIX Assoc., 2005, pp. 41–46.

[13] Freescale Semiconductor, *MC146818 – Real-Time Clock Plus RAM (RTC)*, http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC146818.pdf, 1988.

[14] G. Stringham, *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development*. Elsevier, 2009.

[15] Texas Instruments, *Digital Temperature Sensor with 2-Wire Interface*, http://www.nxp.com/documents/user_manual/UM10204.pdf, Sep. 2011.

[16] NXP Semiconductors, *UM10204 – I^2C – bus specification and user manual*, http://www.nxp.com/documents/user_manual/UM10204.pdf, Oct. 2012.

[17] I. Mohor, "Ethernet MAC 10/100 mbps," http://opencores.org/project,ethmac, Jul. 2011.

[18] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th annual Linux Showcase & Conference*, vol. 5. USENIX Assoc., 2001, pp. 18–26.

[19] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *SAC*. ACM, 2010, pp. 2103–2110.

[20] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *CACM*, vol. 54, no. 7, pp. 68–76, Jul. 2011.

[21] C. Cadar and D. R. Engler, "Execution generated test cases: how to make systems code crash itself," in *SPIN*. Springer, 2005, pp. 2–23.

[22] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *CACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010.

[23] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *Int. J. Parallel Program.*, vol. 34, no. 1, pp. 61–91, Feb. 2006.

[24] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *USENIXATC*. USENIX Assoc., 2010, pp. 12–12.

[25] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: a comparative survey," *Int. J. Parallel Program.*, vol. 33, no. 6, pp. 585–611, Dec. 2005.

[26] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Combining software and hardware verification techniques," *FMSD*, vol. 21, no. 3, pp. 251–280, Nov. 2002.

[27] D. Große, U. Kühne, and R. Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," in *GLSVLSI*. ACM, 2006, pp. 43–48.

[28] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz, "Formal hardware/software co-verification by interval property checking with abstraction," in *Design Automation Conference (DAC)*. ACM, 2011, pp. 510–515.

[29] D. Monniaux, "Verification of device drivers and intelligent controllers: a case study," in *EMSOFT*. ACM, 2007, pp. 30–36.

[30] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey, "An automata-theoretic approach to hardware/software co-verification," in *FASE*. Springer, 2010, pp. 248–262.

[31] T. Todman, P. Boehm, and W. Luk, "Verification of streaming hardware and software codesigns," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 147–150.

An SMT Based Method for Optimizing Arithmetic Computations in Embedded Software Code

Hassan Eldib and Chao Wang
Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA
E-mail: {heldib,chaowang}@vt.edu

***Abstract*—We present a new method for optimizing the C/C++ code of embedded control software with the objective of minimizing implementation errors in the linear fixed-point arithmetic computations caused by overflow, underflow, and truncation. Our method relies on the use of an SMT solver to search for alternative implementations that are mathematically equivalent but require a smaller bit-width, or implementations that use the same bit-width but have a larger error-free dynamic range. Our systematic search of the bounded implementation space is based on an inductive synthesis procedure, which guarantees to find a solution as long as such solution exists. Furthermore, the synthesis procedure is applied incrementally to small code regions – one at a time – as opposed to the entire program, which is crucial for scaling the method to programs of realistic size and complexity. We have implemented our method in a software tool based on the Clang/LLVM compiler and the Yices SMT solver. Our experiments, conducted on a set of representative benchmarks from embedded control and DSP applications, show that the method is both effective and efficient in optimizing fixed-point arithmetic computations in embedded software code.**

I. Introduction

Analyzing and optimizing the fixed-point arithmetic computations in embedded control software is crucial to avoid overflow and underflow errors and minimize truncation errors within the designated input range. Implementation errors such as overflow, underflow, and truncation can lead to degradation of the computation results, which in turn may destabilize the entire system. The conventional solution is to carefully estimate the minimum bit-width required by the software code to run in the error-free mode and then choose a microcontroller that matches the minimum bit-width. However, this can be expensive or even infeasible, e.g., when the microcontroller at hand has 16 bits but the code requires 17 bits.

In many cases, it is possible for the developer to manually reorder the arithmetic operations to avoid the overflow and underflow errors and to minimize the truncation errors. However, the process is labor intensive and error prone. In this paper, we present a new compiler assisted code transformation method to automate the process. More specifically, we apply inductive synthesis incrementally to optimize the arithmetic computations so that the code can be safely executed on microcontrollers with a smaller bit-width.

Consider the code in Fig. 1, where all input parameters are assumed to be in the range $[0, 9000]$. A quick analysis of this program shows that, to avoid overflow, the program must be executed on a microcontroller with at least 32 bits. If it were to run on a 16-bit microcontroller, many of the arithmetic operations, e.g., the subtraction in Line 13, would overflow. In this case, a naive solution is to scale down the bit-widths of the overflowing operations by eliminating some of their least significant bits (LSBs). However, it would decrease the dynamic range, ultimately leading to a large accumulative error in the output.

Our method, in contrast, can reduce the minimum bit-width required to run this fixed-point arithmetic computation code without loss in accuracy. It takes the original C code in Fig. 1 and the user-specified ranges of its input parameters, and returns the optimized C code in Fig. 2 as output. Our method guarantees that the two programs are mathematically equivalent but the one in Fig. 2 requires a smaller bit-width. More specifically, the new code can run on a 16-bit microcontroller. Furthermore, our method ensures that the new code does not introduce additional truncation errors.

The optimization in our method is carried out by an SMT solver based *inductive synthesis* procedure, which is customized specifically for efficient handling of fixed-point arithmetic computations. Recent years have seen a renewed interest in applying inductive synthesis techniques to a wide variety of applications (e.g., [1], [2], [3], [4], [5], [6], [7], [8]). However, a naive application of inductive synthesis to our problem would not work, due to the limited scalability and large computational overhead of the synthesis procedure. For example, our experience with the Sketch tool [1] shows that, for synthesizing arbitrary fixed-point arithmetic computations, it does not scale beyond programs with 3-4 lines of code.

Our main contribution in this paper is a new *incremental inductive synthesis* algorithm, where the SMT solver based analysis is carried out only on small code regions of bounded size, one at a time, as opposed to the entire program. This incremental optimization approach allows our method to scale up to programs of practical size and complexity.

Our new method differs from existing methods for optimizing arithmetic computations in embedded software. These existing methods, including recent ones [9], [10], focus primarily on computing the optimal (smallest) bit-widths for the program variables. Instead, our method focuses on re-ordering the arithmetic operations and re-structuring the code, which in turn may lead to reduction in the minimum bit-width. In other words, we are not merely *finding* the minimum bit-width, but also *reducing* it through proper code transformation. Due to the use of an SMT solver based exhaustive search, our method can find the best implementation solution within a bounded search space.

We have implemented our method in a software tool based on the Clang/LLVM compiler framework [11] and the Yices SMT solver [12]. We have evaluated its performance on a representative set of public domain benchmarks collected from embedded control and digital signal processing (DSP) applications. Our results show that the new method can significantly reduce the minimum bit-width required by the program and,

ISBN 978-0-9835678-3-7/13.

```
1: int comp(int A,int B,int H,int E,int D,int F,int K) {
2:  int t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;
3:  t12 = 3 * A;
4:  t10 = t12 + B;
5:  t11 = H << 2;
6:  t9  = t10 + t11;
7:  t6  = t9 >> 3;
8:  t8  = 3 * E;
9:  t7  = t8 + D;
10: t5  = t7 - 16469;
11: t3  = t5 + t6;
12: t4  = 12 * F;
13: t2  = t3 - t4;
14: t1  = t2 >> 2;
15: t0  = t1 + K;
16: return t0;
17:}
```

Fig. 1. The original C program for implementing an embedded controller.

```
1: int comp(int A,int B,int H,int E,int D,int F,int K) {
2:  int t0,t1,t3,t4,t5,t6,t8,t12;
3:  int N1,N2,N3,N4,N5,N6,N7,N9,N10;
4:  t12 = 3 * A;
5:  N6  = H;
6:  N10 = t12 - B;
7:  N9  = N10 >> 1;
8:  N7  = B + N9;
9:  N5  = N7 >> 1;
10: N4  = N5 + N6;
11: t6  = N4 >> 1;
12: t8  = 3 * E;
13: N3  = t8 - 16469;
14: t5  = N3 + D;
15: t3  = t5 + t6;
16: t4  = 12 * F;
17: N2  = t4 >> 2;
18: N1  = t3 >> 2;
19: t1  = N1 - N2;
20: t0  = t1 + K;
21: return t0;
22:}
```

Fig. 2. Optimized C code for implementing the same embedded controller.

alternatively, increase the error-free dynamic range.

To sum up, this paper makes the following contributions:

- We propose the first method for incrementally optimizing the linear fixed-point arithmetic computations in embedded C/C++ code via inductive synthesis to reduce the minimum bit-width and increase the dynamic range.
- We implement the new method in a software tool based on Clang/LLVM and the Yices SMT solver, and demonstrate its effectiveness and scalability on a set of representative embedded control and DSP applications.

The remainder of this paper is organized as follows. In Section II, we illustrate our new method by using an example. We establish the notation in Section III, and then present the overall algorithm in Section IV. We present our inductive synthesis procedure in Section V. The implementation details and experimental results are given in Section VI and Section VII, respectively. We review related work in Section VIII, and finally give our conclusions in Section IX.

II. Motivating Example

We illustrate the overall flow of our method using the example in Fig. 1. The program is intended to be simple for ease of presentation. In the actual evaluation benchmarks, the programs have loops and variables that are assigned more than once. Note that loops in these programs are all bounded, and therefore can be handled by finite unrolling.

Our method takes the program in Fig. 1 and a configuration file that defines the value ranges of all parameters as input, and returns the program in Fig. 2 as output. It starts by parsing the original program and constructing an abstract syntax tree (AST). Each variable in Fig. 1 corresponds to a node in the AST. The root node is the return value of the program. The leaf nodes are the input parameters.

The AST is first traversed *forwardly*, from the parameters to the return value, to compute the value ranges. Each value range is a (min, max) pair for representing the minimum and maximum values of the node, computed using a symbolic range analysis [13]. Then, the AST is traversed *backwardly*, from the return value to the parameters, to identify the list of AST nodes that may overflow or underflow when using a reduced bit-width. For example, the first overflowing node in Fig. 1 is the subtraction in Line 13: although `t3` and `t4` can be represented in 16 bits, the subtraction may produce a value that requires more bits.

For each AST node that may overflow or underflow, we carve out some neighboring nodes to form a *region for optimization*. The region includes the node, its parent node, its child nodes, and optionally, the transitive fan-in and fan-out nodes up to a bounded depth. The region size is limited by the capacity of the inductive synthesis procedure. For the subtraction in Line 13, if we bound the region size to 2 AST levels, the extracted region would include the right-shift in Line 14, which is the parent node.

The region is then subjected to an inductive synthesis procedure, which generates an equivalent region that does not overflow or underflow. For Line 13 in Fig. 1, the extracted region and the new region are shown side by side as follows:

```
t2 = t3 - t4;                    N2 = t4 >> 2;
t1 = t2 >> 2;          -->       N1 = t3 >> 2;
...                              t1 = N1 - N2;
```

That is, instead of applying right-shift to the operands after subtraction, it applies right-shift first. Because of this, the new region needs a smaller bit-width to avoid overflow.

However, the above new region is not always better because it may introduce additional truncation errors. Consider `t3 = 2, t4 = -2` as a test case. We have `(t3 - t4) >> 2 = 1` and `(t3 >> 2 - t4 >> 2) = 0`, meaning that the new region may lose precision if the two least significant bits (LSBs) of `t3,t4` are not zero. An integral part of our new synthesis method is to make sure that the new region does not introduce additional truncation errors. More specifically, we perform a truncation error margin analysis to identify, for each AST node, the number of LSBs that are immaterial in deciding the final output. For Line 13, this analysis would reveal that the LSBs of `t3` and `t4` do not affect the value of the final output.

Since the new region is strictly better, the original AST is updated by replacing the extracted region with the new region. After that, our method continues to identify the next node that may overflow or underflow. The entire procedure terminates when it is no longer possible to optimize any further.

In the remainder of this section, we provide a more detailed description of the subsequent optimization steps.

After optimizing the subtraction in Line 13, the next AST node that may overflow is in Line 10. The extracted region and the new region are shown side by side as follows:

```
t7 = t8 + D;                     N3 = t8 - 16469;
```

```
t5 = t7 - 16469;        -->        t5 = N3 + D;
```

Our analysis shows that variables `t8, D` and constant `16469` all have zero truncation error margins. The new region does not introduce any additional truncation error. Therefore, the original AST is updated with the new region.

The next AST node that may overflow is in Line 6. The extracted region and the new region are shown as follows:

```
t9 = t10 + t11;                  N6 = t11 >> 2;
t6 = t9 >> 3;                    N5 = t10 >> 2;
...                       -->    N4 = N5 + N6;
...                              t6 = N4 >> 1;
```

The truncation error margins are 2 for `t10` and 2 for `t11`. Therefore, the truncation error margin for `t9` is 2, meaning that the two LSBs may be ignored. Since the new region is strictly more accurate, the original AST is again updated with the new region.

The next AST node that may overflow is in Line 4. The extracted region and the new region are shown as follows:

```
t10 = t12 + B;                   N10 = t12 - B;
N5  = t10 >>2;                   N9  = N10 >> 1;
...                       -->    N7  = B + N9;
...                              N5  = N7 >> 1;
```

Notice that this extracted region consists of a node that is the result of a previous optimization step. The truncation error margins are 0 for `t12` and 0 for `B`. The new code region does not suffer from the same truncation error that would be introduced by `N5 = (B>>2 + t12>>2)`, because the truncation error is not amplified while being propagated to the final result. Instead, it is compensated by the addition of `B`.

The last node that may overflow is in Line 5 of Fig. 1. The extracted region and the new region are shown as follows:

```
t11 = H << 2;
N6  = t11 >> 2;          -->        N6 = H;
```

By now, all arithmetic operations that may overflow are optimized. The new program in Fig. 2 can run on a 16-bit microcontroller while still maintaining the same accuracy as the original program running on a 32-bit microcontroller. Another way to look at it is that if the optimized code were to be executed on the original 32-bit microcontroller, it would have a significantly larger dynamic range.

III. Preliminaries

A. Fixed-point Notations

We follow [14] to represent the fixed-point type by a tuple $\langle s, N, m\rangle$, where s indicates whether it is signed or unsigned (1 for signed and 0 for unsigned), N is the total number of bits or the *bit-width*, and m is the number of bits representing the fractional part. The number of bits representing the integer part is $n = (N - m)$. Different variables and constants in the original program are allowed to have different bit representations, but all of them should have the same bit-width N.

Signed numbers are represented in the standard *two's complement* form. For an N-bit number α, which is represented by bit-vector $x_{N-1}\ x_{N-1}\ ...\ x_0$, its value is defined as follows:

$$\alpha = \frac{1}{2^m} \times \left(-2^{N-1} x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i \right) ,$$

where x_i is the value of the i^{th} bit. The value of α lies in the range $[-2^n, 2^n - 2^{-m}]$. If a number to be represented exceeds the maximum value, there will be an *overflow*. If a number to be represented is less than the minimum value, there will be an *underflow*. If the number to be represented requires more designated fractional bits than m, there will be a *truncation error*. The maximum error caused by truncation is 2^{-m}.

We define the *step* of a variable or a constant as the number of consecutive LSBs that always have the value zero. For example, the number 1024 has a *step* 9, meaning that nine of the LSBs are zero. On the other hand, the number 3 has a *step* 0. During the optimization process, they will be used to compute the truncation error margin (the LSBs whose values can be ignored). Our method will leverage the truncation error margins to obtain the best possible optimization results.

B. Intermediate Representation

We use Clang/LLVM to construct an intermediate representation (IR) for the input program. Since the standard C language cannot explicitly represent fixed-point arithmetic operations, we use a combination of the integer C program representation and a separate configuration file, which defines the fixed-point types of all program variables. More specifically, we scale each fixed-point constant (other than the ones used in *shift* operations) to an integer by using the scaling factor 2^m. For example, a constant with the value of 2.5 will be represented as 10, together with $m = 2$, since $2.5 * 2^2 = 10$.

After each multiplication, a *shift-right* is added to normalize the result so as to match the fixed-point type for the result. For example, $x = c \times z$, where variables x and z and constant c all have the fixed-point type $\langle 1, 8, 3\rangle$, would be represented as $x = (c \times z) >> 3$. Our implementation currently supports linear fixed-point arithmetic only; therefore, we do not consider the multiplication of two variables.

Although there is no inherent difficulty in our method for handling non-linear arithmetic, we focus on linear arithmetic for two reasons. First, the benchmarks used in our experiments are all linear. Second, we have not evaluated the efficiency of SMT solvers in handling non-linear arithmetic operations. Therefore, we leave the handling of nonlinear arithmetic for future work.

For each multiplication, we also assign an *accumulate flag*, which can be set by the user to indicate whether the microcontroller has the capability of temporally storing the multiplication result into two registers, which effectively doubles the bit-width of the registers. Many real-world microcontrollers have been designed in this way. Continuing with the same example $x = (c \times z) >> 3$, if the *accumulate flag* is set to 1 by the user, the multiplication node will not be checked for overflow and underflow. Only after the right-shift, will the final result be checked for overflow and underflow.

For all the other operations (`+, -, >>, <<`), we do not rewrite the default IR representation and do not allow the user to set the *accumulate flag*, because most of the microcontrollers do not have double sized registers to temporarily store the results of these operations.

IV. The Overall Algorithm

The overall flow of our method in shown in Algorithm 1. The input includes the original program and the value ranges

of all the parameter variables. First, we invoke COMPUTERANGES to compute the value ranges of all non-leaf AST nodes. Then, we invoke COMPUTEIGNOREBITS to compute the truncation error margins (LSBs whose values can be ignored) for all AST nodes. Finally, we compute the bit-width ($bw1$) required by the original program to run within the given input range.

Algorithm 1 Optimizing the program within its input range.

```
1: OPTIMIZEPROGRAM (prog, p_ranges) {
2:   ranges ← COMPUTERANGES(prog, p_ranges);
3:   ig_bits ← COMPUTEIGNOREBITS(prog);
4:   bw1 ← COMPUTMINBITWIDTH(prog, ranges);
5:   while (true) {
6:     bw2 ← bw1 − 1;
7:     for each (Node n ∈ prog that may overflow or underflow) {
8:       reg ← EXTRACTREGION(prog, n);
9:       new_reg ← SYNTHESIZE(reg, bw1, bw2, ranges, ig_bits);
10:      if (new_reg does not exist) break;
11:      REPLACEREGION(prog, reg, new_reg);
12:    }
13:    bw1 ← bw2;
14:  }
15:  return prog;
16: }
```

After the bit-width of the original program ($bw1$) is determined, we enter the while-loop to iteratively optimize the program. In each iteration, we try to reduce the bit-width from $bw1$ to $bw2$. The loop terminates as soon as a call to the inductive synthesis procedure fails to return the new region.

Within each loop iteration, we search for all nodes that may overflow or underflow when the new bit-width ($bw2$) is used. We process these nodes in a breadth-first search (BFS) order, i.e., from the return value of the program to the parameter variables. For each node, we invoke EXTRACTREGION to extract a neighboring region and then invoke the inductive synthesis procedure. If successful, the inductive synthesis procedure would return a new region, which is mathematically equivalent to the extracted region but would not overflow or underflow. It also ensures that the new region would not introduce additional truncation error. After the new region is found, we use it to replace the extracted region in the program.

A. Region for Optimization

The size of the extracted *region* affects both the effectiveness and the computational overhead of the inductive synthesis procedure. The minimum extracted region should include the erroneous node and its parent node. Since we follow the BFS order, the parent node must have no overflow or underflow since it is already tested negative or optimized. Since in the original program, the parent operation restores the overflowed value created in the overflowing node back to the normal operation range, when the parent node is included in the region, it is more likely to find an alternative implementation that is more accurate than the extracted region.

In general, a larger extracted region allows for more opportunity to find a suitable new region. The maximum extracted region – if it were not for the limited capability of the SMT solver – would be the entire input program. This is equivalent to applying inductive synthesis tools such as Sketch [1], [2] to the entire program, provided that the fixed-point arithmetic optimization problem is modeled in the Sketch input language. In practice, however, such a monolithic optimization approach seldom works. Indeed, our experience with the Sketch tool shows that it cannot scale beyond arbitrary fixed-point arithmetic computation code of 2-3 lines.

Therefore, in addition to implementing our customized inductive synthesis procedure, which can efficiently handle fixed-point arithmetic computations, we also bound the size of the extracted region so that inductive synthesis is applied only in the context of incremental optimization. More specifically, the extracted region is bounded to an AST with at most 5 node levels, which represents up to 63 AST nodes.

B. Truncation Error Margin

We compute the *step* and the *ignore bits* for all AST nodes recursively. First, we determine the *step* of each leaf node based on the definition in Section III. In general, the *step* may originate from a *shift-left* operation, a *step* in a parameter variable, or a *step* in a constant. We compute the step of each internal AST node as follows:

- $step(x * y) = step(x) + step(y)$;
- $step(x + y) = min(step(x), step(y))$;
- $step(x - y) = min(step(x), step(y))$;
- $step(x << c) = step(x) + c$;
- $step(x >> c) = max(step(x) - c, 0)$.

The *ignore bits* are those consecutive LSBs that can be ignored during the optimization process. If these bits are truncated in the new region, for example, no error will occur in its output. By taking into account these bits in the optimization process, we are able to synthesis more compact new regions.

To clarify this, consider the example in Fig. 3, where the extracted region is shown inside the dotted box. We start by analyzing the AST to determine the *step* of each node. For the purpose of optimizing the extracted region, we need to know the *step* of the region's inputs, which are the nodes labeled as a and b. Due to the shift-left operations, the *step* of a is 4, while the *step* of b is 3. Considering these *step* values, we determine that, when optimizing the extracted region, we have a "credit" of 3 bits to ignore. In other words, we have the freedom to truncate up to 3 consecutive LSBs of the two inputs (a and b) without decreasing the accuracy of the result. Because of this, we are able to synthesize the new region as shown in Fig. 4.

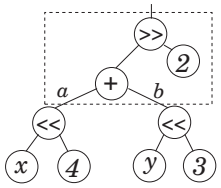


Fig. 3. The extracted region.

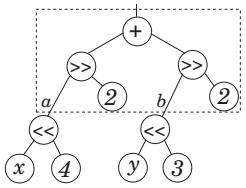


Fig. 4. The synthesized region.

Notice that, even if we do not consider the ignore bits, our method can still synthesize a new region to remove the overflowing node in the above example. However, in such case, the extracted region would have to be larger. That is, the extracted region would need to include all the AST nodes in Fig. 3. The synthesized new region would include all the AST nodes in Fig. 4. However, this would also lead to a significantly longer synthesis time.

V. The Inductive Synthesis Procedure

At the high level, our inductive synthesis procedure consists of two steps: (1) run a set of test cases on the extracted region, and based on the results, generate a new region that is equivalent to the extracted region at least for the set of test cases; (2) check if the two regions are equivalent in the full input range. If they are not equivalent, block this region (bad solution) and try again.

Algorithm 2 shows the pseudo code of our synthesis procedure, which computes a new region (new_reg) of bit-width $bw2$, such that it is equivalent to the original region (reg) of bit-width $bw1$, under the value ranges specified in $ranges$ while considering the truncation error margins specified in ig_bits. The procedure starts by initializing *blockedRegions* and *testSet* to empty sets, where *testSet* consists of the test cases used for inductively generating (guessing) a new region, and *blockedRegions* consists of the previously explored regions that fail the equivalence check. The procedure initializes the *size* of the new region to 1, and then enters the while-loop to iteratively search for a new region of increasingly larger size. When *size* exceeds a predetermined bound, we have proved that no solution exists in this search space.

Subroutine GENREGION uses an SMT solver to inductively generate a new region, based on the test examples in *testSet* and the already explored regions in *blockedRegions*. Subroutine COMPDIFF formally checks the equivalence of the extracted region (reg) and the new region (new_r), and returns a concrete test if they are not equivalent.

Algorithm 2 Inductively synthesizing the new code region.

```
1: SYNTHESIZE (reg, bw1, bw2, ranges, ig_bits) {
2:   blockedRegions ← { };
3:   testSet ← { };
4:   size ← 1;
5:   while (size < MAX_REGION_SIZE) {
6:    new_r ← GENREGION(reg, bw1, bw2, size, blockedRegions, testSet);
7:     if (new_r exists) {
8:       test ← COMPDIFF(reg, new_r, bw1, bw2, ranges, ig_bits);
9:       if (test exists) {
10:        blockedRegions ← blockedRegions ∪{new_r};
11:        testSet ← testSet ∪{test};
12:      }
13:      else
14:        return new_r;
15:    }
16:    else
17:      size ← size + 1;
18:  }
19:  return no_solution;
20: }
```

A. Constructing the New Region Skeleton

First, we generate a *skeleton* of the new region, which is a generalized AST capable of representing any linear arithmetic equation up to a bounded size. In this AST, each leaf node is either a constant or any of the set of input variables of the extracted *region*. Each internal node is any of the linear arithmetic operations (`*, +, -, >>, <<`). The root node is the result of the arithmetic computation and should compute the same result as the output node in the extracted region. Fig. 5 shows an example *skeleton* of 7 AST nodes. Here, *Op* represents any binary arithmetic operator and $V|C$ represents a leaf node (either a variable or a constant).

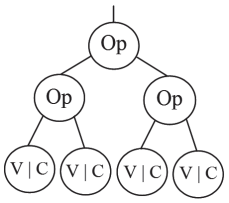


Fig. 5. Skeleton of 7 AST nodes.

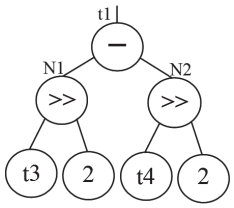


Fig. 6. Synthesized new region.

For each AST node in the *skeleton*, we assign an auxiliary variable called the *selector*, whose value determines the node type. For example, a leaf node (`LNode1`), which may be variable `V1`, variable `V2`, or constant `C1`, is represented as follows:

```
((LNode1 == V1) && (sel1 == 0) ||
 (LNode1 == V2) && (sel1 == 1) ||
 (LNode1 == C1) && (sel1 == 2))
```

where the integer value of selector variable `sel1` ranges from 0 to 2. Similarly, a generalized internal node (`INode3`), which may be an addition or a subtraction of `LNode1` and `LNode2`, is represented as follows:

```
((INode3 == LNode1+LNode2) && (sel2 == 0) ||
 (INode3 == LNode1-LNode2) && (sel2 == 1))
```

where the integer value of selector variable `sel2` ranges from 0 to 1. The actual node types in the *skeleton* are determined later, when we encode the skeleton into an SMT formula, and then call the SMT solver to find a set of suitable values for all these selector variables.

B. Inductively Generating the New Region

To generate the new region, we need a representative set of test cases for the extracted region. These are test values for the input variables of the region, and should include both the corner cases and the intermediate values. Since the arithmetic computations are linear, we construct the corner cases by including the minimum and maximum values of all input variables as defined in *ranges*. Additional test values are generated by taking semi-equidistant intermediate values between values in the corner cases.

We create an SMT formula Φ such that Φ is satisfiable iff the resulting new region – induced by a satisfying assignment to all *selector* variables – is mathematically equivalent to the extracted region, but does not overflow or underflow.

$$\Phi = \Phi_{reg} \wedge \Phi_{skel} \wedge \Phi_{sameI} \wedge \Phi_{sameO} \wedge \Phi_{tests} \wedge \Phi_{blocked},$$

where the subformulas are defined as follows:

- Extracted region (Φ_{reg}): It encodes the transition relation of the extracted region by using bit-vector arithmetic, where the bit-width is *bw1*.
- New region skeleton (Φ_{skel}): It encodes the transition relation of the skeleton by using bit-vector arithmetic, where the bit-width is *bw2*.
- Same input values (Φ_{sameI}): It asserts that the input variables of the two regions must share the same values.
- Same output value (Φ_{sameO}): It asserts that the output variables of the two regions must have the same value, and there is no overflow or underflow.

- Test cases (Φ_{tests}): It asserts that the input variables must adopt concrete values from the given test cases.
- Blocked solutions ($\Phi_{blocked}$): It asserts that the *selector* variables should not take values that represent any previously explored (bad) solution.

If Φ is unsatisfiable, no solution exists in the bounded search space. In this case, we need to increase the *size* of the *skeleton* and try again. If Φ is satisfiable, we have computed a candidate new region. As an example, consider the first extracted region in Section II. The new region generated from the skeleton in Fig. 5 is shown in Fig. 6.

C. Checking the Equivalence of the Regions

The candidate new region is guaranteed to be equivalent to the extracted region over the given set of test cases. However, they may not be equivalent over the full input range. Therefore, the next step is to formally verify their equivalence over the full input range. Toward this end, we create another SMT formula Ψ, which is satisfiable iff the two regions are *not* equivalent; that is, if there exists a test case that can differentiate them. Formula Ψ is defined as follows:

$$\Psi = \Phi_{reg} \wedge \Phi_{new\_reg} \wedge \Phi_{sameI} \wedge \Phi_{diffO} \wedge \Phi_{ranges} \wedge \Phi_{ig\_bits},$$

where the subformulas are defined as follows:

- New region (Φ_{new_reg}): It encodes the transition relation of the candidate new region in bit-vector arithmetic, where the bit-width is *bw2*.
- Different output values (Φ_{diffO}): It asserts that the output variables of the two regions have different values.
- Value ranges (Φ_{ranges}): It asserts that all input variables should stay within their pre-computed value ranges. We are not interested in checking the equivalence of the two regions outside the designated value ranges.
- Ignore bits (Φ_{ig_bits}): It asserts that the LSBs as specified in the ignore bits should all be set to zero. This allows us to ignore the differences between the two regions for LSBs within the truncation error margins.

If Ψ is unsatisfiable, it means that the two regions are mathematically equivalent within the given input range and under the consideration of the ignore bits.

If Ψ is satisfiable, the candidate new region is not correct. In this case, we add it to the *blockedRegions* and try again. The blocking of an incorrect solution follows the counter-example guided inductive synthesis algorithm [1], [15], where the blocked solutions are encoded as an additional constraint in the SMT formula, by adding an extra pair of extracted *region* and new region *skeleton* with the blocked assignment to *selector* variables. It ensures that, when the SMT solver is invoked again to find a candidate new region, the same solution will not be returned.

VI. Implementation

We have implemented our new method in a software tool for optimizing the C/C++ code of embedded control and DSP applications based on the Clang/LLVM compiler framework [11] and the Yices SMT solver [12]. Our tool has two modes: the whole-program optimization mode and the incremental optimization mode. The two modes differ only in the size bound imposed on the extracted region.

When the bound is set to an arbitrarily large number, our tool runs in the whole-program optimization mode. This makes it somewhat comparable to the popular inductive synthesis tool called Sketch [1], [15], provided that our new region *skeleton* is carefully modeled in the Sketch input language, with the *selector* variables defining the "integer holes" for Sketch to fill. Before implementing our own inductive synthesis procedure, we have explored this approach. However, it turns out to be not scalable: synthesizing a new region with a size bound of more than 2 would cause Sketch to quickly run out of the 4 GB memory. We believe that there are two reasons for this. First, the performance of Sketch is not optimized for handling arbitrary combinations of linear fixed-point arithmetic computations. Second, inductive synthesis, in general, may not be able to scale up to arbitrarily large arithmetic computation programs.

Due to the scalability problem encountered by using Sketch, we have implemented our own inductive synthesis procedure directly using the Yices SMT solver, which is designed for efficient handling of fixed-point arithmetic operations, e.g., by designing SMT encoding schemes for exploiting the AST structures encountered in this type of applications. Our experimental evaluation shows that the new procedure is significantly more efficient than Sketch. Instead of a size bound of 2, it now can routinely optimize the *skeleton* with a size bound of 5 (representing up to 63 AST nodes). Nevertheless, this improvement alone is not sufficient for supporting the whole-program optimization.

Instead, we propose an incremental optimization method that applies inductive synthesis only to individual regions of a bounded size. More specifically, we have set the maximum bound for *shift-right* and *shift-left* operations to 4, and the maximum level of AST nodes in the new region skeleton to 5. By incrementally optimizing one extracted region at a time, our method is able to avoid the scalability bottleneck imposed by the SMT solver, and therefore can be applied to programs of practical size and complexity.

VII. Evaluation

We have evaluated our tool on a set of public domain benchmark examples. The experiments are designed to answer the following three questions:

- How much can our method reduce the minimum bit-width required for the program to run in the given input range?
- How much can our method increase the dynamic range of the program for the given bit-width?
- If both the original and the optimized programs are forced to run with a reduced bit-width, what is the difference between their fixed-point specific implementation errors?

A. Benchmarks

Our benchmark includes a set of public domain C programs for embedded control and DSP applications. They come from various sources including papers, textbooks, and the output of code generation tools. The sizes of the programs range from 21 lines of code (LoC) to 131 lines, with an average LoC of 79. The number of fixed-point arithmetic operations on average is 58. For the kind of cyber-physical systems (CPS) software targeted by our new method, these are programs of realistic size and complexity.

TABLE I
STATISTICS OF THE BENCHMARK C PROGRAMS.

Name of the Benchmark	Line of Code	Arithmetic Operations
Sobel Image filter (3x3)	42	28
Bicycle controller	37	27
Locomotive controller	42	38
IDCT (N=8)	131	114
Control. Impl.	21	8
Diff. image filter (5x5)	131	77
FFT (N=8) (no DC component)	112	82
IFFT (N=8)	112	90

Table I shows the statistics of our benchmarks. The first test case, taken from [16], is a 3x3 Sobel digital filter that is widely used in image processing applications. The second test case, taken from [10], is a bicycle controller optimally synthesized for a custom-designed microprocessor with double-sized internal registers. The third test case is a locomotive controller generated by using Fixed Point Advisor and Real Time Workshop of the Matlab toolkit [17]. The fourth test case, taken from [18], is an inverse discrete cosine transform (IDCT), which is widely used in mobile communication and image compression applications. The fifth test case is the fixed-point version of a control rule implementation from [17]. The sixth test case is a 5x5 kernel sized difference image filter [19]. The seventh test case is a fast Fourier transform (FFT) implementation, where the floating-point version was taken from [20] and then converted to fixed-point, by changing all `double` variables into `int` variables without modifying or reordering any of its instructions. The eighth test case is the inverse fast Fourier transform (IFFT) for test case 7. None of the benchmarks was modified from their original forms in any way to give performance advantage to our method.

All experiments were conducted on a machine with a 3.4 GHz Intel i7-2600 CPU, 3.3GB of RAM, and 32-bit Linux.

B. Results

First, we show that there is a significant increase in the input/output range from the original program to the optimized program, when they both use the original bit-width. Table II shows the results (data on the output range are similar, and therefore are omitted for brevity). Column 1 shows the name of the benchmark. Columns 2 and 3 show the input (output) ranges of the original program and the optimized program, respectively. Column 4 shows the percentage of the range increase. The increase in input (output) range spans from 0% to 1515%, with an average of 307% or a median of 72%. The increase is due to the removal of the overflowing and underflowing nodes in the original program. As a result, the output range is also increased. Together, they lead to a significant increase in the dynamic range of the entire application.

Second, we show that there is a significant decrease in the minimum bit-width required for the program to run without overflow/underflow errors for the given input range. The experimental results are shown in Table III. Column 1 is the name of the benchmark. Column 2 is the minimum bit-width of the original program to avoid overflow and underflow, and Column 3 is the average bit-width for all program variables. Column 4 is the minimum bit-width of the new program to avoid overflow and underflow, and Column 5 is the average bit-width for all program variables.

TABLE II
INCREASE IN THE OVERFLOW/UNDERFLOW FREE INPUT RANGE.

benchmark	bit	original	optimized	%
Sobel Image	32	[0, 16320]	[-65536, 49152]	602
Bicycle	32	[-3.4*10^8, 3.4*10^8]	[-1.0*10^9, 1.0*10^9]	194
Locomotive	64	[-8.7*10^{18}, 8.7*10^{18}]	[-9.2*10^{18}, 9.2*10^{18}]	5
IDCT	32	[0, 1.5*10^6]	[0, 2.1*10^6]	40
Controller	32	In1 [0, 5.0*10^8]	In1 [-0, 6.6*10^8]	32
		In2 [-5.0*10^8, 0]	In2 [-6.6*10^8, 0]	32
		In3 [-5.0*10^8, 0]	In3 [-6.6*10^8, 0]	32
Diff. Image	32	[0, 1.3*10^8]	[0, 2.1*10^9]	1515
FFT (N=8)	32	[0, 32736]	[0, 32736]	0
IFFT (N=8)	32	[0, 2.6*10^8]	[0, 5.3*10^8]	103

TABLE III
INCREASE IN THE MINIMUM AND AVERAGE BIT-WIDTHS.

Name of Benchmark	Original (bit-width)		Optimized (bit-width)	
	Minimum	Average	Minimum	Average
Sobel image filter (3x3)	17	10.26	15	6.67
Bicycle controller	18	14.47	16	14.16
Locomotive controller	33	29.41	32	29.32
IDCT (N=8)	20	16.29	19	16.38
Control. Impl.	17	15	16	14.67
Diff. image filter (5x5)	17	11.11	13	8.09
FFT (N=8)	18	7.32	16	6.95
IFFT (N=8)	17	7.11	16	7.26

Our results show that the bit-width reduction spans from 1 bit to 4 bits. Consider the Sobel Image filter as an example. The minimum bit-width required to run the original program is 17 bits. After optimization, it is reduced to 15 bits. This is significant, because now the code can be executed on a 16-bit microcontroller instead of a 32-bit microcontroller, which is often significantly cheaper.

To further illustrate the benefit of our new method, consider the maximum error bound in a scaled-down version of the original program in order to downgrade the hardware from 32-bit to 16-bit, or from 64-bit to 32-bit. Table IV shows the comparison between the optimized program and a scaled-down version of the original program. Column 1 is the name of the benchmark. Column 2 is the scaling level. Columns 3 and 4 are the maximum relative errors of the original program and the optimized program, respectively. Our results show that the optimized programs have smaller errors in all test cases.

We also show, in Table V, the statistics of running our optimization method. Column 1 is the name of the benchmark. Column 2 is the number of lines optimized by the incremental inductive synthesis procedure in the original program. Column 3 is the total execution time by our method. The data show that, by using incremental synthesis, we have kept the overall runtime down. In fact, it is no longer directly dependent on the

TABLE IV
DECREASE IN THE MAXIMUM RELATIVE ERROR.

Benchmark	Scaling	Error original	Error optimized
Sobel Image filter (3x3)	32-b $\rightarrow$ 16-b	$3.1 * 10^{-2}$	0.0
Bicycle controller	32-b $\rightarrow$ 16-b	$3.5 * 10^{-4}$	$2.0 * 10^{-4}$
Locomotive controller	64-b $\rightarrow$ 32-b	$2.9 * 10^{-8}$	$1.5 * 10^{-9}$
IDCT (N=8)	32-b $\rightarrow$ 16-b	$9.2 * 10^{-3}$	$1.8 * 10^{-5}$
Control. Impl.	32-b $\rightarrow$ 16-b	$5.2 * 10^{-4}$	$2.9 * 10^{-4}$
Diff. image filter (5x5)	32-b $\rightarrow$ 16-b	$1.2 * 10^{-2}$	$2.5 * 10^{-3}$
FFT (N=8)	32-b $\rightarrow$ 16-b	$8.1 * 10^{-2}$	$4.4 * 10^{-3}$
IFFT (N=8)	32-b $\rightarrow$ 16-b	$8.4 * 10^{-2}$	$3.2 * 10^{-}2$

TABLE V
STATISTICS OF THE INCREMENTAL OPTIMIZATION PROCESS.

Name of the Benchmark	Num. Optimized Lines	Total Time
Sobel Image filter (3x3)	22	2s
Bicycle controller	2	5s
Locomotive controller	1	5m 41s
IDCT (N=8)	3	2.7s
Control. Impl.	1	46s
Diff. image filter (5x5)	23	10s
FFT (N=8)	14	1m9s
IFFT (N=8)	1	4s

program size, but more on the number of extracted regions and the time spent on optimizing each region. For *Locomotive*, the SMT solver took a longer time because of its larger original bit-width (64-bit) – the other examples are all 32-bit.

VIII. Related Work

Our new method incrementally optimizes the fixed-point arithmetic computations in an embedded software program with the objective of reducing the minimum bit-width through code transformation, without changing the computational accuracy. The core synthesis routine in our method follows the same counter-example guided inductive program synthesis paradigm pioneered by Sketch [1], [2]. However, our method is significantly different in that it has an implementation that is designed for more efficiently handle linear fixed-point arithmetic computations. Furthermore, we apply inductive synthesis incrementally to regions of a bounded size, one at a time, as opposed to the entire program.

Gulwani *et al.* [5] propose a method for synthesizing bit-vector programs from a linear reference code by leveraging a set of user defined library functions. Their method does not use incremental inductive synthesis, and the largest synthesized code reported in their paper has 16 lines of code, for which their tool takes over 45 minutes. Jha *et al.* [3] use the same symbolic encoding as in [5] but replace the logical specification of the desired program by an input-output oracle.

The SCIDUCTION tool implemented by Jha [9] can automatically synthesize a fixed-point arithmetic program from the floating-point arithmetic code. However, the focus of this tool is solely on *finding* the smallest possible bit-width and *choosing* the best fixed-point representation for each program variable. They have not attempted to change the code structure or synthesize completely new code for the purpose of *reducing* the minimum bit-width.

Another closely related work is the linear fixed-point optimization method proposed in [10], which relies on using a Mixed Integer Linear Programming (MILP) solver to minimize the error bound by changing the fixed-point representation of the program. Again, their method can only optimize the bit-vector representations of the program variables, but do not change the structure of the original code or synthesize new completely new code in order to reduce the bit-width.

Our method is also related to superoptimization in modern compilers [21], [22], [23], which perform exhaustive search in the space of valid instruction sequences to optimize performance-critical inner loops. However, they typically cannot be used to increase the dynamic range, or minimize the bit-width, of fixed-point arithmetic computations.

IX. Conclusions

We have presented a new method for incrementally optimizing the linear fixed-point arithmetic computations of an embedded software program via code transformation to reduce the required bit-width and to increase the dynamic range. Our method is based on judicious application of an SMT solver based inductive synthesis procedure to code regions of bounded size. We have implemented our method in a software tool and evaluated it on a set of representative embedded programs. Our results show that the new method can significantly reduce the bit-width and handle programs of realistic size and complexity.

X. Acknowledgments

This work is supported in part by the NSF grant CNS-1128903 and the ONR grant N00014-13-1-0527.

References

[1] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *PLDI*, 2005, pp. 281–294.
[2] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *PLDI*, 2008, pp. 136–148.
[3] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010, pp. 215–224.
[4] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011, pp. 317–330.
[5] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.
[6] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *PLDI*, 2011, pp. 317–328.
[7] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI*, 2012, pp. 275–286.
[8] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *International Conference on Computer Aided Verification*, 2012, pp. 634–651.
[9] S. K. Jha, "Towards automated system synthesis using sciduction," Ph.D. dissertation, UC Berkeley, Nov 2011.
[10] M. Rupak, I. Saha, and M. Zamani, "Synthesis of minimal-error control software," in *ACM international conference on Embedded software*, 2012, pp. 123–132.
[11] C. Lattner and V. Adve, "The LLVM Instruction Set and Compilation Strategy," CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2002-2292, Aug 2002.
[12] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 81–94.
[13] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *PLDI*, 2000, pp. 182–195.
[14] R. Yates, *Fixed-point arithmetic: An introduction*. Digital Signal Labs, Technical Reference, 2013.
[15] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006, pp. 404–415.
[16] S. Qureshi, *Embedded Image Processing on the TMS320C6000 DSP*. Springer, 2005.
[17] A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *ACM international conference on Embedded software*, 2010, pp. 9–18.
[18] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for c and c++ based digital signal processing programs," in *IEEE Trans. Circuits and Systems II*, vol. 45, no. 11, 1998, pp. 1455–1464.
[19] W. Burger and M. Burge, *Digital Image Processing*. Springer, 2008.
[20] J. Xiong, J. R. Johnson, R. W. Johnson, and D. A. Padua, "Spl: A language and compiler for dsp algorithms," in *PLDI*, 2001, pp. 298–308.
[21] R. Joshi, G. Nelson, and K. H. Randall, "Denali: A goal-directed superoptimizer," in *PLDI*, 2002, pp. 304–314.
[22] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *ASPLOS*, 2006, pp. 394–403.
[23] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ASPLOS*, 2013, pp. 305–316.

Verifying Periodic Programs with Priority Inheritance Locks

Sagar Chaki
Software Engineering Institute
Email: chaki@sei.cmu.edu

Arie Gurfinkel
Software Engineering Institute
Email: arie@cmu.edu

Ofer Strichman
Technion
Email: ofers@ie.technion.ac.il

***Abstract*—Periodic real-time programs are ubiquitous: they control robots, radars, medical equipment, etc. They consist of a set of tasks, each of which executes (in a separate thread) a specific job, periodically. A common synchronization mechanism for such programs is via Priority Inheritance Protocol (PIP) locks. PIP locks have low programming overhead, but cause deadlocks if used incorrectly. We address the problem of verifying safety and deadlock freedom of such programs. Our approach is based on sequentialization – converting the periodic program to an equivalent (non-deterministic) sequential program, and verifying it with a model checker. Our algorithm, called PIPVERIF, is iterative and optimal – it terminates after sequentializing with the smallest number of rounds required to either find a counterexample, or prove the program safe and deadlock-free. We implemented PIPVERIF and validated it on a number of examples derived from a robot controller.**

I. INTRODUCTION

Periodic programs are widely used to control safety-critical systems. They consist of multiple tasks, each performing a specific job (typically, by invoking a function) periodically. Each task runs in its own thread of execution. Thus, periodic programs are inherently concurrent. They have, however, unique characteristics. First, the arrival and maximum processing times of jobs are known *a priory*. Second, each thread has a *unique* and – other than the issue of locks discussed below – *fixed* priority. Hence both the inherent non-determinism of job arrival and the complexity of the scheduling policy (e.g., one that depends on a job's time in the queue) that characterize general concurrent software, are absent for periodic programs. Periodic programs are designed to be correct only under these restrictions. Therefore, verifying them against a completely non-deterministic scheduler (as common with general concurrent software) is too imprecise.

To address this challenge, we developed [1][2] an approach for *time-bounded verification* of periodic programs. Our approach leverages the restrictions on scheduling and job arrival mentioned above. Given a periodic program $\mathcal{C}$ and a time bound t, we verify that $\mathcal{C}$ does not violate a safety property φ when executed for time t from an initial state I. We assume that t, φ and I are user-specified. Our scheduler model is not completely non-deterministic. It preserves relative ordering of jobs and priorities, while abstracting away concrete time. It is thus sound for properties that depend only event ordering, and not the exact times at which events occur. Note that restricting execution time (as opposed to, say, number of context switches [3]) is more natural for a periodic program since time maps directly to the program's execution state. For example, the software that deploys an airbag in a car completes in a fixed amount of time, and therefore, during verification, we are interested in bugs that occur within that time limit only.

Periodic programs use locks for synchronization. However, such locks must prevent priority inversion [4], whereby a thread is blocked by another with lower priority. A priority inversion almost caused the failure of the 1997 PathFinder mission [5]. To this end, several locking protocols have been proposed in literature [4]. Real-time operating systems [6] typically support two versions – the Priority Ceiling Protocol (PCP) lock and the Priority Inheritance Protocol (PIP) lock. Both types of locks prevent priority inversion. The PCP lock eliminates deadlocks as well, but requires additional programming effort. In contrast, the PIP lock is easier to use but leads to deadlock if used incorrectly. In earlier work [1][2] we explored the time-bounded verification of periodic programs with PCP locks. In this paper, we deal with PIP locks.

We use the sequentialization paradigm proposed by Lal and Reps [7], and build on our earlier work on sequentializing periodic programs without PIP locks [2]. In [2], every execution of the periodic program is partitioned logically into *rounds*. During sequentialization, we first fix the total number of rounds. Next, each job (i.e., the periodic execution of a task) is *scheduled*, i.e., assigned a starting and an ending round. Jobs are then executed in order of increasing priority and starting time. Before executing each statement, a job non-deterministically *context switches*, i.e., jumps to a higher round, thereby modeling preemption. Finally, constraints are used to ensure that jobs are appropriately scheduled (e.g., a job never starts while another with higher priority is executing), properly preempted (e.g., a job never preempts another with higher priority), and that rounds are consistent (the value of each shared variable at the end of a round equals its value at the beginning of the next round).

In the context of periodic programs with PIP locks, existing sequentialization approaches [7][2] are inadequate for several reasons. First, the priority of a thread changes dynamically. More importantly, due to priority inheritance, it is possible for the priority of a thread to change even while the thread itself is suspended. Second, an exact bound on the number of rounds needed to account for all possible executions cannot be determined efficiently. Finally, periodic programs with PIP locks

ISBN 978-0-9835678-3-7/13.

can deadlock. However, the existing sequentialization-based deadlock detection algorithm for concurrent programs [8] do not work with priorities, because it requires that every deadlock have a wait-free counterexample. This is not true when priorities are involved (see Sec. IV for more details). Against this background, we make the following contributions.

First, we present an iterative algorithm, called PIPVERIF, for verifying a time-bounded periodic program with PIP locks. PIPVERIF maintains a number R of rounds, starting with R = the total number of jobs. In each iteration, it first checks for counterexamples to safety with R rounds. If such a counterexample, is detected, PIPVERIF terminates with UNSAFE. Otherwise, it checks for the presence of executions with more than R rounds. If there are no such executions, PIPVERIF terminates with SAFE. Otherwise, it increments R and continues with the next iteration. PIPVERIF is optimal – it terminates with the smallest R required to either find a counterexample, or prove the program safe.

Second, we extend PIPVERIF to detect deadlocks. To this end, the sequential program that we generate maintains the transitive closure of the Task-Resource Graph (TRG) [9] in an incremental manner. A node of the TRG represents either a task or a PIP lock. An edge from a task t to a lock l means that the currently executing job of t is blocked trying to acquire l. Similarly, an edge from a lock l to a task t means that l is held by the currently executing job of t. Detecting a deadlock state is equivalent to detecting that the TRG is cyclic.

Finally, we implement PIPVERIF by extending REKH [2]. We validate our tool, called REKPIP, on a set of examples derived from the controller of a LEGO Mindstorms robot. In each case, REKPIP produces the correct result, either proving the program SAFE, producing a counterexample for a user-specified safety property, or detecting a deadlock. These results indicate that our approach is feasible. Our tools and benchmarks are available at http://www.andrew.cmu.edu/user/arieg/Rek/start-rekpip.cde.tar.gz.

It is important to note that assuming a nondeterministic scheduler, as done by virtually the entire literature on concurrent program verification, makes these verification methods *inherently incomplete* even when the execution is bounded, simply because in the real system the scheduler is *not* nondeterministic. The current line of work is therefore the first to present, to the best of our knowledge, a sound and *complete* – relative to the time-bound, and for properties that only depend on the ordering of events – verification method for a (particular type of) a concurrent program. It is also the first empirically validated verification method for periodic programs with PIP locks. Given the popularity of such systems and their criticality, preventing deadlocks and guaranteeing their safety properties is no doubt an important problem.

The rest of this paper is organized as follows. In Section II, we present basic concepts and definitions. In Section III, we present PIPVERIF in details. In Section IV, we survey related work. Finally, we present our implementation, benchmarks, and results in Section V, and conclude in Section VI.

II. PRELIMINARIES

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is the priority, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – the period, C – the worst case execution time (WCET) of T, and A, called the release time, is the time at which the task is first enabled[1]. A *periodic program* (PP) is a set of tasks. In this paper, we consider a N-task PP $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$, where $\tau_i = \langle I_i, T_i, P_i, C_i, A_i\rangle$. We assume that: (i) for simplicity, $I_i = i$; (ii) execution times are positive, i.e., $C_i > 0$; (iii) priorities are *rate-monotonic* [10] and distinct – tasks with smaller period have higher priority; and (iv) $\mathcal{C}$ is schedulable. Let RT_i be the response time of τ_i (i.e., the maximum time taken by any job of τ_i to complete) computed via Rate Monotonic Schedulability [11] analysis.

Bounding Time and Jobs. We verify $\mathcal{C}$ assuming that it executes for one "hyper-period" $\mathcal{H}$ [11], where $\mathcal{H}$ is the least common multiple of $\{P_0, \ldots, P_{N-1}\}$. We refer to the resulting time-bounded program as $\mathcal{C}_{\mathcal{H}}$. We also assume that the first job of each task finishes before its period, i.e.,

$$\forall 0 \leq i < N \centerdot\ A_i + RT_i \leq P_i\ . \tag{1}$$

Under this restriction, the number of jobs of task τ_i that executes in $\mathcal{C}_{\mathcal{H}}$ is:$J_i = \frac{\mathcal{H}}{P_i}$. The semantics of $\mathcal{C}_{\mathcal{H}}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i{:=}0; \textbf{while}(k_i < J_i \wedge \textsc{Wait}(\tau_i, k_i))\ (T_i; k_i{:=}k_i{+}1)\ . \tag{2}$$

where $\|$ is preemptive priority-sensitive interleaving (the CPU is always given to the enabled task with the highest priority, preempting the currently executing task if necessary), $k_i \in \mathbb{N}$ is a counter and $\textsc{Wait}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i + k_i \times P_i$, and otherwise blocks until time $A_i + k_i \times P_i$ and then returns TRUE. In the rest of the paper, for simplicity and brevity, we write $\mathcal{C}$ to mean $\mathcal{C}_{\mathcal{H}}$.

Synchronization. We assume that tasks synchronize via priority inheritance protocol (PIP) locks [4]. Trying to acquire a PIP lock l involves one of two possibilities. If l is available, it is taken and execution proceeds normally. If the lock is unavailable, the current thread (executing, e.g., task τ) is blocked and the (suspended) thread holding l inherits τ's priority and hence resumes execution. The resumed thread drops back to its previous (i.e., prior to resumption) priority as soon as it releases l, and goes back to being suspended. Note that PIP locks cause blocking, and therefore deadlocks, if used improperly.

Example 1: Consider the task set in Fig. 1(a). A partial schedule (up to time 9) for these values is shown in Fig. 1(b). At time 0, τ_0 starts and acquires l_1. At time 1, τ_1 preempts τ_0 and acquires l_2. At time 2, τ_2 preempts τ_1. At time 3, τ_2 tries to acquire lock l_2 and gets blocked. At this point, τ_1 inherits τ_2's priority (i.e., 2) and resumes execution. At time 4, τ_1 tries to acquire lock l_1 and gets blocked. At this point, τ_0 inherits τ_1's priority (i.e., 2) and resumes execution. At time 5, τ_0 releases lock l_1. The inherited priority of τ_0 drops back to its previous

[1]We assume that time is given in some fixed unit (e.g., milliseconds).

(a)

Task	I_i	A_i	C_i	P_i
τ_2	2	2	2	10
τ_1	1	1	4	20
τ_0	0	0	3	40

(b)

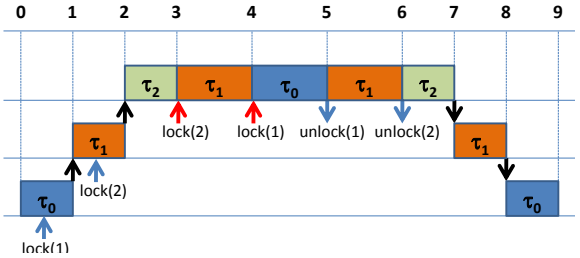


Fig. 1. (a) Three tasks from Example 1; (b) A schedule of the three tasks.

priority, viz., 0, and it is preempted by τ_1 which grabs lock l_1. At time 6, τ_1 releases lock l_2. The inherited priority of τ_1 drops to 1, and it is preempted by τ_2 which grabs lock l_2. At time 7, τ_2 releases lock l_2 and terminates, and τ_1 resumes execution. At time 8, τ_1 releases lock l_1 and terminates, and τ_0 resumes execution. At time 9, τ_0 terminates.

We write $\mathsf{J}(\tau, k)$ to denote the k-th job (i.e., the job at the k-th position) of task τ. Thus, the set of all jobs of $\mathcal{C}$ is:

$$\mathsf{J} = \bigcup_{0 \le i < N} \{\mathsf{J}(\tau_i, k) \mid 0 \le k < J_i\} \,. \tag{3}$$

Job Ordering. Consider a job $j = \mathsf{J}(\tau_i, k_i)$. Recall that A_i, P_i and RT_i are, respectively, the release time, period, and response time of τ_i. Then, the arrival time of j is $A(j) = A_i + k_i \times P_i$, and the departure time of j is $D(j) = A(j) + RT_i$. Since we assume that $RT > 0$, we know that $A(j) < D(j)$. Let $\pi(j) = i$, i.e., the priority of τ_i. We define three ordering relations (developed in our earlier work [2]) on jobs.

Definition 1: The relations $\lhd$, $\uparrow$ and $\sqsubset$ are defined as:

$$\begin{aligned} j_1 \lhd j_2 &\iff (\pi(j_1) \le \pi(j_2) \wedge D(j_1) \le A(j_2)) \vee \\ &\quad\ (\pi(j_1) > \pi(j_2) \wedge A(j_1) \le A(j_2)) \\ j_1 \uparrow j_2 &\iff \pi(j_1) < \pi(j_2) \wedge A(j_1) < A(j_2) < D(j_1) \\ j_1 \sqsubset j_2 &\iff (A(j_1) < A(j_2)) \vee \\ &\quad\ (A(j_1) = A(j_2) \wedge \pi(j_1) > \pi(j_2)) \end{aligned}$$

Note that $j_1 \sqsubset j_2$ means that either j_1 always completes before j_2, or it is possible for j_1 to be preempted by j_2. Also, $\sqsubset$ is a total strict ordering since it is a lexicographic ordering by (arrival time, -priority).

Execution. Let $x \bullet y$ be the concatenation of x and y. An execution ρ is a finite sequence of actions where an action is either a job getting blocked (b), or an assertion being violated (a). Note that, for any $k \ge 0$, b^k is the set of executions with k blocks and $b^k \bullet a$ is the set of executions that end with assertion violations and have k blocks. The semantics of a periodic program $\mathcal{C}$, denoted by $[\![\mathcal{C}]\!]$, is a set of executions. Let $[\![\mathring{\mathcal{C}}]\!]$ be the prefix-closure of $[\![\mathcal{C}]\!]$, i.e.,

$$[\![\mathring{\mathcal{C}}]\!] = \{x \mid \exists y \in \{b, a\}^* \,.\ x \bullet y \in [\![\mathcal{C}]\!]\}$$

We say that $\mathcal{C}$ is unsafe iff $\exists k \ge 0 \,.\ b^k \bullet a \in [\![\mathcal{C}]\!]$.

Algorithm 1 The overall verification algorithm. Function VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE if $\mathcal{C}$ has a counterexample (CEX) with R rounds, INCROUNDS if $\mathcal{C}$ has no R round CEXs, but has legal executions with more than R rounds, and SAFE otherwise, i.e., if $\mathcal{C}$ has no CEXs with R or more rounds.

```
1:  function PIPVERIF(C)
2:      R := |J|
3:      loop
4:          x := VERIFROUNDS(C, R)
5:          if x = INCROUNDS then R := R + 1
6:          else return x

7:  function VERIFROUNDS(C, R)
8:      if [[S_a(C, R)]] ≠ ∅ then return UNSAFE
9:      if [[S_b(C, R)]] ≠ ∅ then return INCROUNDS
10:     else return SAFE
```

III. Job-Bounded Verification

Our verification algorithm PIPVERIF uses the idea that any execution ρ of $\mathcal{C}$ is partitioned into scheduling *rounds* as follows: (a) ρ begins in round 0, and (b) a round ends and a new one begins every time a job ends (i.e., the last instruction of some task body is executed) or gets blocked when trying to acquire a lock.

Example 2: The bounded execution in Fig. 1(b) is partitioned into 5 rounds as follows: round 0 is the time interval $[0, 3]$ – when τ_2 gets blocked trying to acquire lock l_2, round 1 is $[3, 4]$ – when τ_1 gets blocked trying to acquire lock l_1, round 2 is $[4, 7]$ – the end of the first job of τ_2, round 3 is $[7, 8]$, and round 4 is $[8, 9]$.

Since the number of rounds that an execution is partitioned into depends on the number of times a job gets blocked, different executions have different number of rounds. More specifically, the execution b^k or $b^k \bullet a$ has exactly $|\mathsf{J}| + k$ rounds. For soundness, PIPVERIF must therefore use a sufficiently large number of rounds during sequentialization. To this end, PIPVERIF starts with a small number of rounds (specifically, $|\mathsf{J}|$) and iteratively increases it till either a real error is detected, or we prove that all executions have been accounted for.

Algorithm 1 shows the pseudo-code of PIPVERIF. Note that, in each iteration, it invokes VERIFROUNDS($\mathcal{C}, R$) to check if:

1) $\mathcal{C}$ has a counterexample with R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE.
2) $\mathcal{C}$ has no counterexample with R rounds, but has legal executions with more than R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns INCROUNDS.
3) $\mathcal{C}$ has no legal executions with more than R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns SAFE.

Correctness of PIPVERIF. PIPVERIF is correct because it explores all legal executions of the program and only terminates when a real counterexample is detected (i.e., if VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE) or when it proves that no more legal executions remain to be explored (i.e., if VERIFROUNDS($\mathcal{C}, R$) returns SAFE).

A. How VERIFROUNDS Works

Recall that VERIFROUNDS$(\mathcal{C}, R)$ must satisfy the following specification:

- if $b^{R-|\mathsf{J}|} \bullet a \in [\![\mathcal{C}]\!]$ then VERIFROUNDS$(\mathcal{C}, R)$ = UNSAFE
- else if $\forall k > R - |\mathsf{J}| \,.\, \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset$ then VERIFROUNDS$(\mathcal{C}, R)$ = SAFE
- else VERIFROUNDS$(\mathcal{C}, R)$ = INCROUNDS

Consider the pseudo-code of VERIFROUNDS (see Alg. 1). First (line 8), it checks if $b^{R-|\mathsf{J}|} \bullet a \in [\![\mathcal{C}]\!]$. To this end, it constructs a sequential program $\mathcal{S}_a(\mathcal{C}, R)$ such that:

$$[\![\mathcal{S}_a(\mathcal{C}, R)]\!] = \emptyset \iff b^{R-|\mathsf{J}|} \bullet a \notin [\![\mathcal{C}]\!] \tag{4}$$

It then checks if $[\![\mathcal{S}_a(\mathcal{C}, R)]\!] = \emptyset$ using a model checker for sequential programs. Next, to prove that:

$$\forall k > R - |\mathsf{J}| \,.\, \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset$$

it relies on the following observation:

$$\forall k > R - |\mathsf{J}| \,.\, \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset \iff b^{R+1-|\mathsf{J}|} \notin [\![\mathring{\mathcal{C}}]\!]$$

Therefore (line 9), it constructs a sequential program $\mathcal{S}_b(\mathcal{C}, R)$ such that:

$$[\![\mathcal{S}_b(\mathcal{C}, R)]\!] = \emptyset \iff b^{R+1-|\mathsf{J}|} \notin [\![\mathring{\mathcal{C}}]\!] \tag{5}$$

and checks whether $[\![\mathcal{S}_b(\mathcal{C}, R)]\!] = \emptyset$ via a model checker for sequential programs. Finally, if both the previous checks fail, it returns SAFE (line 10). In terms of complexity, the construction of $\mathcal{S}_a(\mathcal{C}, R)$ and $\mathcal{S}_b(\mathcal{C}, R)$ are each polynomial in the size of $\mathcal{C}$. The complexity of the subsequent model checking depends on the tool used (e.g., NP for CBMC).

B. Constructing $\mathcal{S}_a(\mathcal{C}, R)$

$\mathcal{S}_a(\mathcal{C}, R)$ reduces the bounded concurrent execution of $\mathcal{C}$ into a sequential execution with R rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, jobs are executed sequentially, in the order $\sqsubset$ defined by Defn. 1. For each global variable g, we guess the initial value of g at the beginning of each round at the start of $\mathcal{S}_a(\mathcal{C}, R)$. At the end of $\mathcal{S}_a(\mathcal{C}, R)$, we ensure that the guessed value of g at the beginning of each round equals its final value at the end of the previous round. In addition, $\mathcal{S}_a(\mathcal{C}, R)$ encodes the inherited priority of jobs and an exception mechanism to detect assertion violations and deadlocks. We now describe these in more detail.

Inherited Priority. Every job $j = \mathsf{J}(\tau, k)$ has a static *base priority* $\pi_b(j)$, which is the priority of the corresponding task τ. In addition, j also has an *inherited priority* $\pi_i(j)$, which changes dynamically as locks are acquired and released. Specifically, at any instant, $\pi_i(j)$ is the maximum of $\pi_b(j)$, and the inherited priorities of all jobs that are blocked on a lock held by j. Note that $\pi_i(j)$ is a global property – it depends not only on the state of j but also on the states of other jobs. The scheduler always executes the non-blocked job with the highest (possibly inherited) priority. Thus, $\mathcal{S}_a(\mathcal{C}, R)$ must keep track of the inherited priorities of jobs to encode PIP locks.

Task-Resource Graph. To compute the inherited priorities of jobs, $\mathcal{S}_a(\mathcal{C}, R)$ encodes the transitive closure of the "task resource graph" [9] (TRG) of the program. The TRG Γ is a dynamic data structure. Its nodes are either tasks or PIP locks. However, its edges depend on the program's execution state. Specifically, an edge from a task t to a lock l means that the currently executing job of t is blocked trying to acquire l. Similarly, an edge from a lock l to a task t means that l is held by the currently executing job of t. Since a job can be blocked on at most one lock at a time, and since a PIP lock can be held by at most one job at a time, a periodic program falls under the category of Single-Resource Model [9] system. For such systems, it is known that Γ is a forest, unless the program's execution state has (two or more) deadlocked tasks [9].

The value of $\pi_i(j)$ is computed from Γ as follows. Let Γ^* denote the transitive closure of Γ, i.e., $(x, y) \in \Gamma^*$ iff there is a path from x to y in Γ. Then,

$$\pi_i(j) = MAX(\{\pi_b(j') \mid (j', j) \in \Gamma^*\}) \,.$$

Thus, if $j = \mathsf{J}(\tau, k)$, then $\pi_i(j)$ is the maximum of the priorities of all tasks that reach τ (including τ itself) in Γ^*. $\mathcal{S}_a(\mathcal{C}, R)$ uses this fact to maintain Γ^* in an online manner – updating it as soon as Γ changes – and compute $\pi_i(j)$ on demand.

Detecting Assertion Violations. In order to model program termination due to an assertion violation, $\mathcal{S}_a(\mathcal{C}, R)$ uses an *exception* mechanism. We use a distinguished global flag to indicate the occurrence of an assertion violation. The flag is initially set to FALSE. Whenever an assertion violation is detected, the corresponding job sets a global flag and exits. All jobs starting (or resuming) in the future check the flag, find it to be set, and also exit. Finally, the flag is used to ensure that $\mathcal{S}_a(\mathcal{C}, R)$ only has a legal execution if $\mathcal{C}$ has an execution with an assertion violation.

Detecting Deadlocks. A deadlock occurs in $\mathcal{C}$ iff its TRG Γ becomes cyclic [9]. More specifically, the deadlocked tasks are exactly the ones whose nodes belong to a cycle in Γ. Therefore, $\mathcal{S}_a(\mathcal{C}, R)$ looks for cycles in Γ whenever a job gets blocked trying to acquire a lock. Since $\mathcal{S}_a(\mathcal{C}, R)$ maintains Γ^* in an online manner, a cycle created in Γ by the addition of an edge is detected in constant time. If a cycle is detected, $\mathcal{S}_a(\mathcal{C}, R)$ uses the exception mechanism described above to indicate an error and abort program execution.

C. Construction of $\mathcal{S}_a(\mathcal{C}, R)$

The structure of $\mathcal{S}_a(\mathcal{C}, R)$ is given by the pseudo-code in Alg. 2 and Alg. 3. Note that $\alpha(e)$ terminates all executions where e evaluates to false. We first describe the global variables of $\mathcal{S}_a(\mathcal{C}, R)$, followed by its functions.

Global Variables of $\mathcal{S}_a(\mathcal{C}, R)$. Recall that $\mathcal{S}_a(\mathcal{C}, R)$ executes the jobs of $\mathcal{C}$ in the order $\sqsubset$ defined by Defn. 1. Each job j is assigned a starting and an ending round during scheduling – these are stored in $start[j]$ and $end[j]$, respectively. Variable rnd stores the current round in which a job is executing. Variable $\mathtt{B}[r]$ indicates whether a job running at round r is allowed to block. Variable $e[r]$ indicates if an exception has been thrown in round r. Variable $\mathtt{P}[r]$ indicates the priority at which the system is executing at round r – this equals the

Algorithm 2 The structure of $\mathcal{S}_a(\mathcal{C}, R)$. Notation: T = set of all tasks; L = set of all PIP locks; J = set of all jobs; G = set of global variables of $\mathcal{C}$; i_g = initial value of g according to $\mathcal{C}$; '$*$' = non-deterministic value; $\alpha() = \text{assume}()$.

var $rnd, start[], end[], \mathtt{B}[], e[], v_e[], \mathtt{P}[], v_{\mathtt{P}}[], \mathtt{S}[][], v_{\mathtt{S}}[][], \mathtt{T}[][][], v_{\mathtt{T}}[][][], \mathtt{L}[][][], v_{\mathtt{L}}[][][]$ $\quad \forall g \in \mathsf{G}$ **.** **var** $g[], v_g[]$

1: **function** MAIN( )
2: INITGLOBS(); HYPPER(); CHECKASSUMPS()
3: **function** INITGLOBS( )
4: $e[0] := 0$; $\forall l \in \mathsf{L}$ **.** $\mathtt{S}[l][0] := -1$
5: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}$ **.** $\mathtt{T}[t_1][t_2][0] := 0$
6: $\forall t \in \mathsf{T}, l \in \mathsf{L}$ **.** $\mathtt{L}[t][l][0] := 0$
7: $\forall g \in \mathsf{G}$ **.** $g[0] := i_g$
8: $\forall r \in [1, R)$ **.** $e[r] := v_e[r] := *; \mathtt{P}[r] := v_{\mathtt{P}}[r] := *$
9: $\forall l \in \mathsf{L}, r \in [1, R)$ **.** $\mathtt{S}[l][r] := v_{\mathtt{S}}[l][r] := *$
10: $\forall t_1, t_2 \in \mathsf{T}, r \in [1, R)$ **.** $\mathtt{T}[t_1][t_2][r] := v_{\mathtt{T}}[t_1][t_2][r] := *$
11: $\forall t \in \mathsf{T}, l \in \mathsf{L}, r \in [1, R)$ **.** $\mathtt{L}[t][l][r] := v_{\mathtt{L}}[t][l][r] := *$
12: $\forall g \in \mathsf{G}, r \in [1, R)$ **.** $g[r] := v_g[r] := *$
13: **function** HYPPER( )
14: SCHEDULEJOBS()
let $j_0 \sqsubset j_1 \sqsubset \ldots j_{|\mathsf{J}|-1}$ *be the job ordering from Defn. 1*
15: RUNJOB(j_0); ...; RUNJOB($j_{|\mathsf{J}|-1}$)
16: **function** RUNJOB(Job j)
17: $rnd := start[j]; o := \mathtt{P}[rnd]; \mathtt{P}[rnd] := \pi_b(j)$
18: **if** $e[rnd] = 0$ **then** $\hat{T}(j)$
19: CS(j); $\mathtt{P}[rnd] := o$; $\alpha(rnd = end[j])$
20: **function** $\hat{T}$(Job j)
let $\sigma \equiv$ **if** $e[rnd] = 1$ **then return**
$\hat{T}$ is obtained from T_t by replacing each 'lock(l)' with:
21: CS(j); σ; LOCK(l, j); σ
each 'unlock(l)' with: CS(j); σ; UNLOCK(l, j)
each 'assert(x)' with:
23: CS(j); σ; **if** $\neg x$ **then** ABORT(j); **return**
and each statement 'st' with:
24: CS(j); σ; $st[g \leftarrow g[rnd]]$
25: **function** CHECKASSUMPS( )
26: **for** $r \in [0, R-1)$ **do** *//let* $r' = r + 1$
27: $\alpha(e[r] = e[r'])$; $\alpha(\mathtt{P}[r] = v_{\mathtt{P}}[r'])$
28: $\forall l \in \mathsf{L}$ **.** $\alpha(\mathtt{S}[l][r] = v_{\mathtt{S}}[l][r'])$
29: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}$ **.** $\alpha(\mathtt{T}[t_1][t_2][r] = v_{\mathtt{T}}[t_1][t_2][r'])$
30: $\forall t \in \mathsf{T}, l \in \mathsf{L}$ **.** $\alpha(\mathtt{L}[t][l][r] = v_{\mathtt{L}}[t][l][r'])$
31: $\forall g \in \mathsf{G}$ **.** $\alpha(g[r] = v_g[r'])$
32: $\forall r \in [0, R)$ **.** $\alpha(\mathtt{B}[r] = 0); \alpha(e[R-1] = 1)$
33: **function** ABORT(Job $j = \mathsf{J}(\tau, k)$)
34: $e[rnd] := 1$
35: $\forall l \in \mathsf{L}$ **.** $\mathtt{S}[l][rnd] = \tau \implies$ UNLOCK(l, j)

(possibly inherited) priority of the currently executing job. For each global variable g of $\mathcal{C}$, variable $g[r]$ indicates its value in round r. The *prophecy* variables $v_e[r]$, $v_{\mathtt{P}}[r]$ and $v_g[r]$ indicate the guessed initial values of $e[r]$, $\mathtt{P}[r]$ and $g[r]$, respectively. The values of $e[r]$, $\mathtt{P}[r]$ and $g[r]$ are updated by the jobs executing in round r only.

Arrays $\mathtt{S}$, $\mathtt{T}$ and $\mathtt{L}$ encode the state of the PIP locks and the transitive closure Γ^* of the TRG. Specifically, $\mathtt{S}[l][r]$ is the priority of the task holding lock l at round r. If l is free at round r, then $\mathtt{S}[l][r] = -1$. Since a task's priority equals its id, we use a task and its priority interchangably. For every pair of tasks (t_1, t_2), $\mathtt{T}[t_1][t_2][r] = 1$ iff $(t_1, t_2) \in \Gamma^*$ at round r. For every task t and lock l, $\mathtt{L}[t][l][r] = 1$ iff $(t, l) \in \Gamma^*$ at round r. Prophecy variables $v_{\mathtt{S}}[l][r]$, $v_{\mathtt{T}}[t_1][t_2][r]$ and $v_{\mathtt{L}}[t][l][r]$ record the guessed initial values of $\mathtt{S}[l][r]$, $\mathtt{T}[t_1][t_2][r]$ and $\mathtt{L}[t][l][r]$, respectively. The values of $\mathtt{S}[l][r]$, $\mathtt{T}[t_1][t_2][r]$ and $\mathtt{L}[t][l][r]$ are updated by jobs executing in round r only.

Functions of $\mathcal{S}_a(\mathcal{C}, R)$. The top-level function is MAIN (see Alg. 2). It initializes all global variables by invoking INITGLOBS (line 2), schedules and executes all jobs by invoking HYPPER (line 2), and finally ensures that only legal executions that terminate with an assertion violation or deadlock are allowed by invoking CHECKASSUMPS (line 2).

INITGLOBS (see Alg. 2) initializes all global variables at each round. In particular, for round 0, all globals are initialized (lines 4–7) to their values at the start of the execution of $\mathcal{C}$. For the remaining rounds, they are initialized (lines 8–12) to non-deterministic guessed values. The guessed values are also recorded in the corresponding prophecy variables.

HYPPER (see Alg. 2) first creates a legal schedule for all jobs by invoking SCHEDULEJOBS (line 14) and then executes each job j (line 15) – in the order $\sqsubset$ defined by Defn. 1 – by invoking RUNJOB(j).

In SCHEDULEJOBS (see Alg. 3), line 2 initializes $\mathtt{B}$ to allow jobs to block in all rounds; line 2 also initializes $start$ and end to non-deterministic values; line 3 ensures that $start[j]$ and $end[j]$ are sequential and within legal bounds; line 4 ensures that jobs are properly separated; line 5 ensures that jobs are well-nested – if j_2 preempts j_1, then it finishes before j_1; and line 6 disables job blocks in all rounds in which a job has been scheduled to end.

RUNJOB(j) (see Alg. 2) sets rnd to the round at which j is scheduled to start (line 17), saves the current system priority and then updates it to the base priority of j (line 17), executes a modified version of j but only if no exception has been thrown (line 18), restores the system priority and ensures that j terminates at the appropriate round (line 19).

$\hat{T}(j)$ (see Alg. 2) is identical to the body of j's task, except that it invokes functions LOCK and UNLOCK (shown in Alg. 3) to model the acquiring and releasing of PIP locks (lines 21–22), models assertion violations by invoking ABORT (line 23), and uses variable $g[rnd]$ instead of g (line 24). In addition, $\hat{T}(j)$ increases the value of rnd non-deterministically (by invoking function CS) to model preemption by higher priority jobs prior to each statement. Finally, whenever the value of rnd increases, $\hat{T}(j)$ checks if an exception has been thrown and terminates the job in this case (using the statement σ). Note that rnd can increase only after a call to CS or LOCK.

CHECKASSUMPS (see Alg. 2) ensures that the final value of each global variable at each round is equal to its prophesied

Algorithm 3 The structure of $\mathcal{S}_a(\mathcal{C}, R)$ continued from Alg. 2.

```
1:  function SCHEDULEJOBS( )
2:      ∀r ∈ [0,R) . B[r] := 1; ∀j ∈ J . start[j] = *; end[j] = *
    // Jobs are sequential
3:      ∀i ∈ [0,N) . ∀k ∈ [0,J_i) . α(0 ≤ start[J(i,k)] ≤ end[J(i,k)] < R)
    // Jobs are well-separated
4:      ∀j1 ◁ j2 . α(end[j1] < start[j2]); ∀j1 ↑ j2 . α(start[j1] ≤ start[j2])
    // Jobs are well-nested
5:      ∀j1 ↑ j2 . α(start[j2] ≤ end[j1] ⇒ (start[j2] ≤ end[j2] < end[j1]))
6:      ∀j ∈ J . B[end[j]] = 0
7:  function UNLOCK(int l,Job J(τ,k))
8:      S[l][rnd] := −1; DELLOCKTASK(l,τ)
9:  function ADDLOCKTASK(int l,Task τ)
10:     ∀t ∈ T \ {τ} . T[t][τ][rnd] := (L[t][l][rnd] = 1) ? 1 : T[t][τ][rnd]
11: function DELLOCKTASK(int l,Task τ)
12:     ∀t ∈ T \ {τ} . T[t][τ][rnd] := (L[t][l][rnd] = 1) ? 0 : T[t][τ][rnd]
13: function ADDTASKLOCK(int l,Task τ)
14:     let c(t) ≡ (t = τ ∨ T[t][τ][rnd] = 1)
15:     ∀t ∈ T . L[t][l][rnd] := c(t) ? 1 : L[t][l][rnd]
16:     s := S[l][rnd]; ∀t ∈ T . T[t][s][rnd] := c(t) ? 1 : T[t][s][rnd]
17: function CS(Job j = J(τ,k))
18:     if (*) then return
19:     o := rnd ; rnd := *; α(o < rnd ≤ end[j])
20:     α(P[rnd] = INHERPRIO(τ))
21: function INHERPRIO(Task τ)
22:     return
            MAX({τ} ∪ {t | T[t][τ][rnd] = 1})
23: function UNBLOCK(int l,Job j)
24:     α(B[rnd] = 1); B[rnd] := 0
25:     o := rnd ; rnd := *
26:     α(o < rnd ≤ end[j])
27:     α(P[o] = P[rnd]); α(S[l][rnd] = −1)
28: function LOCK(int l,Job j = J(τ,k))
29:     if S[l][rnd] = −1 then
30:         S[l][rnd] = τ
31:         ADDLOCKTASK(l,τ)
32:     else
33:         if T[S[l][rnd]][τ][rnd] then
34:             ABORT(j); return
35:         ADDTASKLOCK(l,τ)
36:         UNBLOCK(l,j); DELTASKLOCK(l,τ)
37:         if e[rnd] = 1 then return
38:         S[l][rnd] = τ
39:         ADDLOCKTASK(l,τ)
40: function DELTASKLOCK(int l,Task τ)
41:     let c(t) ≡ (t = τ ∨ T[t][τ][rnd] = 1)
42:     ∀t ∈ T . L[t][l][rnd]:=
            c(t) ? 0 : L[t][l][rnd]
```

initial value at the next round (lines 26–31), all rounds have been exhausted by either a job termination or a job block (line 32), and an exception has been thrown (line 32). Line 32 is critical to ensure the property of $\mathcal{S}_a(\mathcal{C}, R)$ given by (4).

ABORT(j) (see Alg. 2) sets the error flag (line 34) and releases all locks held by j (line 35). To release a lock, it invokes UNLOCK (see Alg. 3) which sets the owner of the lock to -1 (line 8) and then removes the edge in the TRG from the current task to the lock (line 8) via DELLOCKTASK.

DELLOCKTASK (see Alg. 3) updates Γ^* by *removing* an edge in Γ from a lock to a task. In contrast, ADDLOCKTASK (see Alg. 3) updates Γ^* by *adding* an edge in Γ from a lock to a task. Similarly, functions ADDTASKLOCK and DELTASKLOCK (see Alg. 3) update Γ^* by, respectively, adding and removing an edge from a task to a lock.

INHERPRIO(τ) (see Alg. 3) returns the inherited priority of the current job task τ at round rnd. It is invoked by CS (see Alg. 3) to ensure (line 20) that whenever a job is preempted, it only resumes at a round where the system priority equals its inherited priority. In addition, CS ensures (line 19) that a job always resumes in a round permitted by the schedule.

LOCK (see Alg. 3) acquires a lock. If the lock is available (line 29) it updates its owner to the current task (line 30) and adds an edge in the TRG (line 31). However, if the lock is held (line 32), it (i) checks for deadlock and aborts if necessary (lines 33–34); (ii) adds an edge in the TRG from the task to lock (line 35); (iii) preempts the task and resumes it in a future round where the lock is available by invoking UNBLOCK (line 36); (iv) deletes the TRG edge from the task to the lock (line 36); (v) checks if an exception has been thrown and aborts if necessary (line 37); (vi) updates the owner of the lock to the current task (line 38); and (vii) adds a TRG edge from the lock to the task (line 39).

UNBLOCK (see Alg. 3) resumes a blocked job in a future round. It ensures that the current round is available for blocking and makes it unavailable for blocking in the future (line 24), and updates the round to a value that is allowed by the schedule (lines 25–26), where the system priority is the same as the current system priority (line 27), and where the lock is available (line 27).

D. Construction of $\mathcal{S}_b(\mathcal{C}, R)$

Recall that $\mathcal{S}_b(\mathcal{C}, R)$ must have the property defined by (5). The structure of $\mathcal{S}_b(\mathcal{C}, R)$ is similar to $\mathcal{S}_a(\mathcal{C}, R)$. The only difference is in $\hat{T}(j)$ and LOCK, which are shown in Alg. 4. Specifically, in $\mathcal{S}_b(\mathcal{C}, R)$: (i) $\hat{T}(j)$ assumes that assertions are never violated (line 4), and (ii) LOCK assumes that whenever a job blocks, then there is no deadlock (line 10), and aborts if there are no available rounds for job blocking (line 11).

IV. Related Work

Several projects use sequentialization [3][8][12] to verify concurrent software. All these approaches assume a non-deterministic scheduler, which is an over-approximation for periodic programs. Of these, our sequentialization is closest to that of Lal and Reps [7] – scheduling is implemented

Algorithm 4 The structure of $\mathcal{S}_b(\mathcal{C}, R)$. We only show functions that are different from $\mathcal{S}_a(\mathcal{C}, R)$.

1: **function** $\hat{T}$(Job j)
   let σ *be the statement* **if** $e[rnd] = 1$ **then return**
   $\hat{T}$ *is obtained from* T_t *by replacing each 'lock(l)' with*:
2: CS$(j); \sigma;$ LOCK$(l, j); \sigma$
3: *each 'unlock(l)' with*: CS$(j); \sigma;$ UNLOCK(l, j)
4: *each 'assert(x)' with*: CS$(j); \sigma; \alpha(x)$
5: *and each statement 'st' with*: CS$(j); \sigma; st[g \leftarrow g[rnd]]$
6: **function** LOCK(int l,Job $j = \mathsf{J}(\tau, k)$)
7: **if** $\mathtt{S}[l][rnd] = -1$ **then**
8: $\mathtt{S}[l][rnd] = \tau;$ ADDLOCKTASK(l, τ)
9: **else**
10: $\alpha(\neg \mathtt{T}[\mathtt{S}[l][rnd]][\tau][rnd])$
11: **if** $\forall r \in [0, R)$ **.** $\neg \mathtt{B}[r]$ **then** ABORT(j); **return**
12: ADDTASKLOCK(l, τ); UNBLOCK(l, j)
13: DELTASKLOCK(l, τ)
14: **if** $e[rnd] = 1$ **then return**
15: $\mathtt{S}[l][rnd] = \tau;$ ADDLOCKTASK(l, τ)

via prophecy variables instead of function calls. Furthermore, our approach limits verification via execution time, instead of context switches [3][8] or some other means.

Kidd et al. [13] also propose to verify real-time software using sequentialization. They model preemptions using function calls, and do not present any tools or experimental results. Their encoding, while useful for obtaining theoretical results, is too imprecise from a practical verification perspective, since it only uses priorities to limit possible preemptions. Indeed, we have shown [2] that the use of job ordering relations (see Defn. 1) eliminates false warnings compared to an approach that uses priorities only. In contrast, we use prophecy variables, following Lal and Reps [7], limit preemptions using job orderings, and validate our approach empirically.

This paper also extends our earlier work on verifying periodic programs [1][2] by handling PIP locks, executions with blockings, and deadlock detection. This requires a more sophisticated sequentialization (e.g., one that encodes the task resource graph), as well as an iterative algorithm to minimize the number of sequentialization rounds.

Lindstrom et al. [14] have used JavaPathfinder to model check real-time Java programs. Their approach is based on discrete event simulation, and does not: (a) rely on WCET, and (b) consider all possible execution times in the range [0,WCET]. Thus, it is not comparable directly to our approach.

Deadlock detection via sequentialization, explored by Rabinovitz and Grumberg [8], assumes that every deadlock has a wait-free counterexample, i.e., an execution where no thread blocks (except at the end where it deadlocks). This is true if the scheduler is non-deterministic (their situation) but not for periodic programs (this work) where priorities are involved.

Task resource graphs have been used for deadlock detection via runtime analysis [15][16] of concurrent software. However, these projects assume a non-deterministic scheduler, and do not use sequentialization. In addition, some of them [16] over-approximate the TRG and report false deadlocks.

File	T	J	Rn	Vars	Cls	SAT	Result
nxt.bug1a.c	29	15	15	1.4M	4.3M	26	UNSAFE
nxt.bug1b.c	58	15	15	2.5M	7.5M	54	UNSAFE
nxt.bug1c.c	61	15	15	2.6M	8.1M	57	UNSAFE
nxt.ok1.c	746	15	17	2.9M	9.0M	714	SAFE
aso.bug1a.c	73	15	15	2.7M	8.3M	68	UNSAFE
aso.bug1b.c	64	15	15	2.6M	8.0M	59	UNSAFE
aso.bug1c.c	33	15	15	1.7M	5.1M	29	UNSAFE
aso.ok1.c	4148	15	19	3.5M	10.9M	4,088	SAFE
aso.bug2a.c	43	15	15	1.6M	4.9M	39	UNSAFE
aso.bug3a.c	48	15	15	1.7M	5.1M	45	UNSAFE
aso.bug3b.c	35	15	15	1.5M	4.6M	32	UNSAFE
aso.bug3c.c	55	15	15	1.6M	4.9M	52	UNSAFE
aso.ok3.c	879	15	16	1.8M	5.5M	866	SAFE
aso.bug4a.c	63	15	15	2.0M	6.1M	58	UNSAFE
aso.bug4b.c	908	15	16	2.1M	6.4M	898	UNSAFE
aso.ok4.c	3047	15	17	2.2M	6.7M	3,027	SAFE

TABLE I
EXPERIMENTAL RESULTS. T = TOTAL TIME (SEC); J = # OF JOBS; RN = # OF ROUNDS AT COMPLETION; VARS = MAX # OF SAT VARIABLES (IN MILLIONS) PRODUCED BY CBMC; CLS = MAX # OF SAT CLAUSES (IN MILLIONS) PRODUCED BY CBMC; SAT = TOTAL TIME USED BY SAT SOLVER.

V. EXPERIMENTS

Our implementation of PIPVERIF, called REKPIP, builds on REKH [2] . The input to REKPIP is a C program containing the task bodies, and annotations to specify priorities, periods, and WCETs. REKPIP uses CIL [17] for sequentialization, and CBMC [18] to verify the resulting C programs. As in other work [7], REKPIP only allows preemption before access of global variables, without losing soundness. We validated REKPIP on several examples derived from the controller of a LEGO Mindstorms robot[2]. All our experiments were done on a Core-i7 machine with four cores (each running at 2.7GHz) and 8GB of RAM. We know of no tool that is comparable directly with REKPIP. Hence, the main purpose of our experiments is to evaluate the feasibility of our approach.

The Controller. The robot controller consists of three tasks (τ_0, τ_1, τ_2) with priorities $(0, 1, 2)$, periods $(48, 24, 4)$, and WCETs $(12, 12, 1)$, respectively. All tasks arrive at time zero. The system is schedulable, the hyper-period $\mathcal{H}$ is 48, and there are 15 jobs in $\mathcal{C}$. The controller must guarantee that when an obstacle is detected, the robot must move backward and not turn, even if the human operator indicates otherwise. This property, called NOCOLLISION, is expressed by an assertion in the controller code. The assertion involves shared variables accessed by multiple tasks. Hence, appropriate mutual exclusion mechanisms must be used to ensure NOCOLLISION.

The Benchmark. The benchmark consists of a set of examples derived from the controller described above. Example `nxt.ok1.c` is derived from the original version of the controller – τ_2 balances and controls the motion (i.e., speed and direction) of the robot, and receives user commands via bluetooth, τ_1 detects obstacles using a sonar sensor, and τ_0 prints log messages. Task τ_0 does not access shared variables

[2]See http://lejos-osek.sourceforge.net/nxtway_gs.htm for more details.

related to NoCollision, while τ_1 and τ_2 ensure NoCollision by using a PIP lock to protect access to the shared variables. The `nxt.bug1*` examples are buggy variations of `nxt.ok1.c` that use the PIP lock inappropriately.

The `aso.*` examples are derived from a modified version of the controller that we constructed by refactoring out the functionality that receives bluetooth commands from τ_2 to τ_0. Example `aso.ok1.c` uses a single PIP lock to protect the shared variables and ensure NoCollision. The `aso.bug1*` series of examples are buggy variations of `aso.ok1.c` that fail to use the PIP lock appropriately.

Example `aso.bug2a.c` tries to ensure NoCollision without requiring the highest priority τ_2 to do any locking or unlocking (thereby ensuring that τ_2 never blocks). Unfortunately, `aso.bug2a.c` is buggy. In contrast, `aso.ok3.c` achieves this goal successfully by combining of a PIP lock and a transaction-based protocol. The `aso.bug3*` series of examples are buggy variations of `aso.ok3.c` that use either the PIP lock, or the transaction-based protocol inappropriately.

Example `aso.ok4.c` improves on `aso.ok4.c` by using two PIP locks for more fine-grained locking. Examples `aso.bug4a.a` and `aso.bug4b.c` are buggy variations of `aso.ok4.c`. The former performs the fine-grained locking incorrectly (one of the tasks releases a lock prematurely), while the latter has a deadlock (tasks τ_0 and τ_1 attempt to acquire the two PIP locks in opposite order).

Results. Table I summarizes our results. pipVerif produces the correct result for all examples. For `nxt.bug1*.c`, columns Rnds and Jobs are always equal, i.e., counterexamples are detected in the first iteration of pipVerif. For `nxt.ok1.c`, two extra rounds are required to prove safety since there are executions with two blockings between (different jobs of) τ_1 and τ_2 via the PIP lock.

For `aso.bug1*.c`, `aso.bug2*.c` and `aso.bug3*.c`, counterexamples are also detected in the first iteration of pipVerif. However, for `aso.ok1.c` and `aso.ok3.c`, pipVerif goes through several iterations, and only proves safety at rounds greater than the number of jobs. In particular, `aso.ok1.c` requires four extra rounds, while `aso.ok3.c` requires only one extra round.

For `aso.bug4b.c`, the deadlock is detected using one extra round. This is because any execution leading to a deadlock must have at least one job blocking. Suppose that two PIP locks are `L0` and `L1`, τ_0 acquires them in the order `(L0, L1)` and τ_1 acquires them in the opposite order. Then for a deadlock to occur, the following situation must occur – τ_0 gets `L0`, τ_0 is preempted by τ_1, τ_1 gets `L1`, τ_1 tries to get `L0` but is blocked, τ_0 inherits τ_1's priority and resumes execution, τ_0 tries to get `L1`, and we have a deadlock.

In general, verifying an `nxt.*` example is faster than verifying a `aso.*` example. We believe that this is due to the factoring out of complex functionality into a separate task (i.e., thread), which results in increased complexity and a larger statespace. The success of RekPip on these benchmarks indicates that our approach is effective, and advances the state-of-the-art in verifying periodic programs with PIP locks.

VI. Conclusion

We presented an iterative algorithm to verify safety and deadlock freedom of periodic programs. Our algorithm is based on sequentialization – reducing the verification of a concurrent program to that of verifying an equivalent (non-deterministic) sequential program. It extends earlier work in this area by handling synchronization via Priority Inheritance Protocol (PIP) locks, and being able to detect deadlocks. It is also optimal in the sense that it terminates with the minimum number of (sequentialization) rounds needed to prove a periodic program safe, or find a counterexample. Empirical validation of our algorithm indicates its feasibility.

Acknowledgment

References

[1] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-Bounded Analysis of Real-Time Systems," in *FMCAD*, 2011.
[2] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, "Compositional Sequentialization of Periodic Programs," in *VMCAI*, 2013.
[3] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *PLDI*, 2004.
[4] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE TC*, vol. 39, no. 9, 1990.
[5] M. Jones, "What really happened on Mars?" http://research.microsoft.com/ mbj/Mars_Pathfinder.
[6] "RTEMS Real Time Operating System," http://www.rtems.org.
[7] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *CAV*, 2008.
[8] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *CAV*, 2005.
[9] C. Shih and J. A. Stankovic, "Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada," University of Massachusetts, Technical report UM-CS-1990-069, 1990.
[10] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *RTSS*, 1989.
[11] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, no. 1, 1973.
[12] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *ICSE*, 2011.
[13] N. Kidd, S. Jagannathan, and J. Vitek, "One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling," in *SPIN*, 2010.
[14] G. Lindstrom, P. C. Mehlitz, and W. Visser, "Model Checking Real Time Java Using Java PathFinder," in *Proc. of ATVA*, 2005.
[15] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," in *SPIN*, 2000.
[16] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *PADTAD*, 2006.
[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *CC*, 2002.
[18] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, 2004.

[3] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN AS-IS BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. DM-0000437

Abstractions for Model Checking SDN Controllers

Divjyot Sethi, Srinivas Narayana, Sharad Malik
Princeton University

***Abstract*—Software defined networks (SDNs) are receiving significant attention in the computer networking community, with increasing adoption by the industry. The key feature of SDNs is a centralized controller which programs the packet forwarding behavior of a distributed underlying network. This centralized view of control—which is absent in traditional networks—opens up opportunities for full formal verification.**

While there is recent research in formal verification of these networks, model checking the controller behavior as it updates the underlying network has only seen limited application. Existing approaches are limited to verifying the controller for a small number of exchanged packets in the network. In this case study, we extend the state of the art by presenting abstractions for model checking controllers for an arbitrarily large number of packets exchanged in the network. We validate the utility of these abstractions through two applications: a learning switch and a stateful firewall.

I. Introduction

Software defined networks (SDNs) (such as ones based on Openflow [1]) have recently received significant attention in the computer networking community, with increasing relevance to and adoption by industry, *e.g.,* [2]. The key feature of SDNs is a centralized controller (*control plane*) which programs a distributed underlying network (*data plane*). While providing a centralized view of control, any bugs in the controller code can be an Achilles heel to the functioning of the entire network [3]. In this case study, we explore abstractions for proving the correctness of controllers using model checking.

Fig. 1 shows an example topology of an SDN. The data plane consists of three hosts H_A, H_B and H_C which exchange packets pkt_1 and pkt_2 with each other via the network switches S_1, S_2 and S_3[1]. These switches consist of ports p_0, p_1 and p_2. Each location in the network (switch ports and hosts) consists of an input and output buffer (referred to as the *data state*). Each switch enqueues packets in its input port buffers, and eventually forwards them to a set of output ports (or drops them) based on the packet processing logic.

The packet processing logic is encoded into the switches in the form of switch *flow tables* (also referred to as the *network state*). Based on the flow table, the switch applies one of the following 3 actions to an incoming packet: (1) it forwards the packet to a set of output ports of that switch, (2) drops the packet or (3) forwards it to the controller. The switch forwards packets to the controller by reporting them as events. The controller is a piece of software which updates switch flow tables through a standard interface (*e.g.,* Openflow [1]), either in response to events forwarded by switches or spontaneously.

[1]Following the approach of Zhang et al. [4], all the network logic like firewalls, routers etc. can be represented using switches.

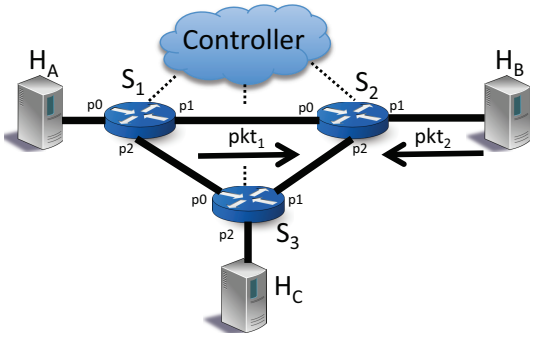


Fig. 1: An example topology.

In this paper, we prove the correctness of the network controller program for a given network topology, and an arbitrarily large number of packets. We focus on *per-packet properties*, which assert the correctness of packet processing in the presence of updates from the controller. Note that these updates may themselves occur in response to events reported due to other packets, *i.e.*, due to *interference* from other packets. Examples of such properties include *no forwarding loop* (*i.e.*, a packet does not loop back to a switch which it has already visited) and *no invalid drop* (*i.e.*, a packet is not dropped due to an invalid controller update to some switch).

Challenges in Verification: The key challenge in model checking SDNs is the state space explosion resulting from the following factors. (1) There can be a large number of packets alive in the network, resulting in a large buffer state—commodity switches can have of the order of ten megabytes of packet buffering per switch (*e.g.,* [5]). Further, these packets create interference for each other by sending events to the controller which then trigger updates to the network state. (2) These packets and the corresponding events can have arbitrary interleavings. (3) The switch flow tables store a mapping from the packet header and the input port to the output port, resulting in a large network state—modern switches can have tens of thousands of flow table entries [6].

Related Work: Existing work in the verification of SDNs exploits the fact that the time between updates to the network state by the controller is much larger than the lifetime of a packet through the network. Thus, the state evolution of the network can be viewed as updates from one network configuration to another via intermediate (*transient*) states, as per the commands from the controller.

Static verification verifies a fixed configuration of the network by using either symbolic simulation [7], by reduction to SAT [4], [8] or, by model checking [9].

Incremental verification approaches extend the static verification approach and incrementally verify the network for

ISBN 978-0-9835678-3-7/13.

all the network configurations [10], [11]. The property may however be violated in the transient stage.

Safe update builds upon the incremental verification approach by guaranteeing that the property under check holds during the transient stage by using specific update protocols [12]. This work is specific to enforcing properties for which the update protocols are designed—more complex properties may not be enforced with this approach.

Dynamic checking seeks to verify the properties on the network in the presence of arbitrary updates from the controller, even when no specific update protocol has been implemented to ensure the property. This is the space of our work—the other works we are aware of in this space are NICE [3] and FlowLog [13]. They use a model checking based approach to successfully find important bugs in controller code. However, they check the controller for a bounded number of exchanged packets. *In this work we extend dynamic checking and scale to an arbitrarily large number of packets.*

Key Contribution: This work addresses the challenges outlined above by first constructing a data state abstraction, and then, a network state abstraction which builds upon the data state abstraction to significantly reduce the model size.

The data state abstraction is based on the standard data type reduction [14] and addresses challenges (1) and (2) by keeping just one packet (*concrete* packet), (pkt^c), in the system and replacing the effect of all the other packets on the network state by *non-deterministically* injecting an arbitrarily large number of *environment packets* (pkt^e). These packets have arbitrary header values, unless constrained by user-added lemmas, and can be injected at any port of any switch. When injected, these environment packets may be forwarded as events to the controller and trigger updates to the network state. Thus, these packets simulate the updates to the network state triggered by an arbitrarily large number of other packets. Further, the environment packets need not traverse the links and occupy data state as they are directly injected at arbitrary ports—this enables abstracting away all the buffers.

Since there is only one alive packet (pkt^c) in the system after the data state abstraction, the network state abstraction exploits this to address challenge (3) by *case splitting* on the source and destination hosts of this packet: the switch flow table can then be abstracted to contain information specific to only these hosts, which are fixed for a concrete packet[2]. This significantly reduces the network state.

Mechanical Construction: The data state abstraction is constructed by adding a special host for injecting environment packets—this host is independent of the controller application under verification. The network state abstraction adds a runtime check to the flow table to only allow updates corresponding to the selected source and destination hosts.

Experimental Setup: We model the controller code to closely resemble the original (typically Python) code in the Murphi language (CMurphi 5.4.6). To verify the controller for a specified topology and property, we use a Murphi model with (1) the controller code, and (2) the switches and hosts connected according to the specified topology with the property specified on it. The abstractions are also implemented on this model.

We demonstrate the utility of the abstraction by verifying *no forwarding loop* for a learning switch application (controller), `Pyswitch` [15] and *no invalid drop* property for a stateful firewall application. We were able to find bugs in the buggy version of these applications and prove correctness for the correct ones for an arbitrary number of packets.

Limitation: While our approach verifies topologies larger than the state of the art for dynamic checking, it cannot scale to realistically large sizes (*e.g.,* data centers). However, in certain cases it is possible to extend controllers proved on smaller topologies to larger ones through topology abstractions [16].

II. Modeling Network Controllers

The network consists of a controller, switches (switch S_i has an id i) with ports (port p_i has an id i) and hosts (host H_i has an id i). Packets traverse the network by hopping from one location to another (unless they get dropped). Each packet consists of a header and payload (data information). The source and destination host id of a packet pkt are denoted by $pkt.src$ and $pkt.dst$, respectively.

When a packet arrives at a switch port, the switch processes the packet in accordance with the *flow table*. For sake of brevity, we assume that the flow table of a switch S (denoted by $S.ft$) matches on the source, destination, and input port (p_i) of the incoming packet and applies a set of actions to the packet—in general our approach extends to cases where it matches other fields as well. Formally, a flow table is a mapping $S.ft : (pkt.src, pkt.dst, p_i) \rightarrow A$, where A is a set containing one or more actions from the following: (1) forward the packet to a set of output ports P_o of the switch (denoted by $Forward\,(P_o)$), (2) drop the packet ($Drop$), or (3) forward it to the controller ($SendToController$). We denote the set of the above three actions by $\mathcal{A}$.

Controller-switch interaction: The controller switch interaction happens in accordance with the Openflow [1] specification: (1) switches report events to the controller which are enqueued in a per-switch event buffer at the controller, and (2) the controller sends commands to the switch which are enqueued in a command buffer at the switch. Following the approach of Foster et al. [17], the relevant events and commands are described below.

Events: The event used in this paper is the $packet\_in(swID, portID, packet)$ event, which tells the controller that $packet$ has arrived at port with id $portID$ of switch with id $swID$.

Controller Commands: The controller can either react to events reported by the switch through event handlers, or spontaneously program the switch. The controller commands can be one of the following: (1) $install(swID,\ match,\ actions)$: this command updates the flow table of switch $swID$ to apply $actions$ (a subset of $\mathcal{A}$) to all packets which match the pattern specified by $match$. The string $match$ is of the form $\{src : H_1, dst : H_2, inport : p_i\}$, *i.e.*, match all incoming packets with $pkt.src = H_1$, $pkt.dst = H_2$, and ingress port p_i at switch $swID$. (2) $send(swID,\ packet,\ action)$ sends the packet to switch $swID$ where $action$ is

[2]Packet rewrites are discussed at the end of §III.

```
packet_in (swID, inport, pkt):
 1: mactable = ctrlState[swID]
 2: mactable[pkt.src] = inport
 3: if (mactable[pkt.dst] != null)
 4:    outport = mactable[pkt.dst]
 5:    if (outport != inport)
 6:       match = {src:pkt.src, dst:pkt.dst, inport:inport}
 7:       action = {Forward (outport)}
 8:       install (swID, match, action)
 9:       send (swID, pkt, action)
10:       return
11: send (swID, pkt, Flood)
12: return
```

Fig. 2: The `Pyswitch` controller algorithm.

applied to it at the switch. Here $action$ is one of $\{Forward, Drop, Flood\}$, where $Flood$ instructs the switch to forward the packet on all its ports except the packet's ingress port.

Property: We verify invariants of the form $\forall pkt : \phi(B_{pkt})$, where ϕ is a propositional logic formula and B_{pkt} is some per-packet book-keeping state. For example, this state can log the packet history, *i.e.*, switches which the packet has visited, in order to detect loops.

Running example (MAC learning switch): We use a layer 2 MAC address learning switch application, `Pyswitch` [15] with topology as shown in Fig. 1, as a running example to describe key concepts. The controller algorithm is shown in Fig. 2. At a high level, for each switch $swID$, the controller learns a mapping (denoted by $ctrlState[swID]$) from host MAC addresses to ports. This allows the switch to forward packets destined to these hosts. As an example, suppose the packet pkt_1, with source H_A and destination H_B, arrives at port p_0 of switch S_2. In case no match exists for the packet, it is matched to a default flow table entry which forwards a $packet\_in$ event to the controller. The `packet_in` event handler learns that the host H_A is reachable through port p_0 on switch S_2 (line 2). In case the port leading to the destination H_B is unknown, the `if` condition on line 3 evaluates to false, and the packet is flooded on all ports, except the incoming port (line 11). However, if the destination is found in $ctrlState$, the flow table is updated to forward all subsequent packets with the same src, dst and $inport$ to p_1 (line 8).

Property: We verify the *no forwarding loop* property for the `Pyswitch` controller. Due to flooding on a topology (Fig. 1) with a loop, the property is violated. However, if the controller only sends packets along a spanning tree (*e.g.,* no packets between port p_2 of S_2 and p_1 of S_3), the property holds.

III. Abstraction

Data state abstraction: As discussed in §I, the data state abstraction exploits the fact that the property under check is a per-packet property: it checks the property on one concrete packet pkt^c, and abstracts away all the other packets. Continuing on the `Pyswitch` example from §II, suppose pkt^e is injected at port p_2 of switch S_2 in Fig. 1, such that $pkt^e.src = H_A$. This leads the switch to send a $packet\_in$ to the controller which updates $ctrlState[S_2][H_A]$ to p_2. Next, suppose the concrete packet pkt^c with source H_B and destination H_A arrives at port p_1 of S_2. The switch sends this packet to the controller, and since $ctrlState[S_2][H_A] = p_2$, the controller commands the switch to forward the packet out port p_2 (line 9 of Fig. 2) to S_3. If there are no matching entries for pkt^c at S_3 and S_1 both in flow tables and $ctrlState$, the packet gets flooded at both switches (line 11). Thus, pkt^c loops back to S_2, which is a violation of the forwarding loop property.

Refinement: Since the header of pkt^e can take arbitrary values, the updates triggered by pkt^e are highly unconstrained. This leads to both scalability bottlenecks as well as functional incorrectness due to *over-abstraction*, *i.e.*, the model exhibiting more behaviors than are realistically expected. We follow the approach of the CMP (CoMPositional) method [18]: the model is iteratively model checked and refined by the user by adding non-interference lemmas in order to constrain pkt^e. The non-interference lemmas we used typically constrain the model according to reachability in the topology. For example a packet with source H_A cannot be injected at port p_1 of S_2, *i.e.*, $((port = p_1)\&(switch = S_2)) \rightarrow (pkt.src \neq H_A)$. (Since these lemmas are application-independent, they were added pre-emptively to mechanically constrain the data state abstraction.) As per the CMP method, these non-interference lemmas do not over-constrain the environment packet pkt^e: they are also model checked by validating them on pkt^c [18].

Network state abstraction: As discussed in §I, the network state abstraction case splits on the source and destination of pkt^c to abstract the flow tables. Suppose we assume that pkt^c has source H_A and destination H_B. Then, for each switch S, the flow table mapping $S.ft$ can be abstracted to $S.ft^{abs}$ where $S.ft^{abs}(pkt^c, p) = S.ft(pkt^c, p)$ for all ports p of the switch, and $S.ft^{abs}(pkt^e, p) = \{SendToController\}$ for all other packets (*i.e.*, pkt^e). The $Forward$ action is not applied to pkt^e as it is non-deterministically injected at all ports.

Packet rewrites: In order to handle packet rewrites, the network state abstraction can be refined by including flow rules for rewritten header values as well. These rules are needed to process the concrete packet when its header is rewritten.

IV. Experiments

We verified two applications: `Pyswitch` and a stateful firewall on a 2.40 GHz Intel Core 2 Quad processor, with 3.74 GB RAM. Murphi source code is available online [19].

MAC learning switch (`Pyswitch`): For the `Pyswitch` application, we verified loop freedom for the star topology shown in Fig. 1, with non-interference lemmas from §III added pre-emptively for scalability. The loop was found in 0.1 sec with 159 states explored. Next, on constraining the topology to forward packets only along the spanning tree, the model checker proved correctness with an arbitrary number of packets exchanged between H_A and H_B in 600s with 1.45M states. We note that model checking did not finish in a day without the abstractions. Finally, we ran a stress test with a larger *fat tree* topology with 20 switches, 16 hosts and 48 links. While model checking did not finish for an arbitrarily large number of packets, it finished in 68352s for the single packet case with network state abstraction.

Stateful firewall: We consider a simple firewall policy which may be used to prevent direct connections from the Internet into an enterprise network, *e.g.,* to implement Network Address Translation (NAT). Fig. 3 shows an enterprise network

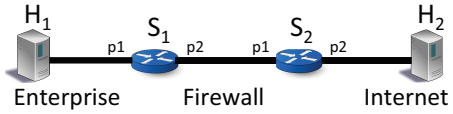


Fig. 3: Two switches acting together as a stateful firewall.

```
1: packet_in (swId, inport, pkt):
2:    if swId = 1 and inport = 1:
3:       match_S1 = {src:pkt.src, dst:pkt.dst, inport:1}
4:       action_S1 = {Forward ({2})}
5:       install (1,match_S1,action_S1)
6:       match_S2 = {src:pkt.dst, dst:pkt.src, inport:2}
7:       action_S2 = {Forward ({1})}
8:       install (2,match_S2,action_S2)
9:    else if swId = 2 and inport = 2:
10:      match = {src:pkt.src, inport:2}
11:      action = {Drop }
12:      install (2,match,action)
```

Fig. 4: Firewall controller for the network shown in Fig. 3.

connected to the Internet via a firewall implemented on two switches S_1 and S_2. The controller (not shown in Fig. 3 for brevity) initializes the *default* behavior of S_1 and S_2 as follows: (1) S_1 sends all incoming traffic at port p_1 to the controller, in addition to forwarding out of p_2, (2) S_1 forwards incoming traffic at p_2 directly to the enterprise host H_1, (3) S_2 sends all incoming traffic at p_1 directly to the Internet, and (4) S_2 forwards incoming traffic at p_2 to the controller. As packets arrive, the controller dynamically updates the switch flow tables to implement the following high-level policy (Fig. 4): (a) All traffic originating from any enterprise host H_1 and destined to any Internet host H_2 is allowed to pass freely. In particular S_1 forwards all traffic to port p_2 bypassing the controller after the first `packet_in` (line 5). (b) Traffic from an Internet host H_2 destined to an enterprise host H_1 is only allowed if the communication was initiated by H_1 first (line 8). (c) If H_2 attempts to communicate with H_1 without prior initiation by H_1, then H_2 is considered malicious and is explicitly blacklisted (line 12).

Verification: We verify the *no invalid drop* property by checking if traffic from Internet host H_2 replying to a request sent by enterprise host H_1 does not get dropped. Due to a race condition between the events reported by S_1 and S_2, the controller erroneously blacklists host H_2. This happens when H_1 sends a first request to H_2. The request goes to S_1 first which forwards it via S_2 to H_2, and also forwards an event e_1 reporting the request to the controller. However, this event gets delayed and in the meantime H_2 replies, and S_2 forwards this to the controller as an event e_2. Since e_2 is processed by the controller before e_1, H_2 is erroneously blacklisted. (Note that events across switches can be processed out of order.) Our approach found this race condition in 0.13 sec with 482 states, without requiring any lemmas.

Fixing the violation: This bug can be fixed by requiring S_1 to wait for the controller before forwarding requests from H_1 to H_2 via S_2. Our approach was able to prove correctness for an arbitrarily large number of packets in 0.19 sec with 613 states, without requiring any lemmas.

V. Conclusion and Ongoing Work

We have presented abstractions for model checking controllers for software defined network applications. These abstractions extend the state of the art by enabling correctness proofs for SDN controllers for an arbitrarily large number of packets and their ensuing controller state updates. As a next step, we plan to explore abstractions to further scale model checking for larger topologies. In particular, since properties are typically violated along a particular path taken by packets in the network, we plan to focus on validating properties for packets taking fixed paths in the topology instead of all possible paths.

Acknowledgment: We thank Jennifer Rexford and Muralidhar Talupur for their helpful ideas and feedback. This work was supported by NSF grant 1111520 and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined WAN," in *SIGCOMM*, 2013.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[4] S. Zhang, S. Malik, and R. McGeer, "Verification of computer switching networks: an overview," ser. ATVA'12. Springer-Verlag, 2012, pp. 1–16.

[5] Broadcom, 2012. [Online]. Available: http://www.broadcom.com/collateral/etp/SBT-ETP100.pdf

[6] IBM G8264 switch, 2012. [Online]. Available: http://www.openflow.org/wp/ibm-switch/

[7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[8] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[9] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated Openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, ser. SafeConfig '10, 2010.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[13] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *Hot topics in Software Defined Networks (HotSDN)*, 2013.

[14] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *CHARME*, 1999.

[15] "PySwitch NOX application," http://nddi.googlecode.com/svn/nox/src/nox/coreapps/examples/pyswitch.py.

[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Symposium on Networked Systems Design and Implementation*, 2013.

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in *International Conference on Functional Programming (ICFP)*, 2011.

[18] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. FMCAD*, 2004.

[19] Murphi source code, [Online] https://github.com/ngsrinivas/sdnverify.

Efficient Modular SAT Solving for IC3

Sam Bayless*, Celina G. Val*, Thomas Ball†, Holger H. Hoos*, Alan J. Hu*

*University of British Columbia, {sbayless, vcelina, hoos, ajh}@cs.ubc.ca
†Microsoft Research, tball@microsoft.com

***Abstract*—We describe an efficient way to compose SAT solvers into chains, while still allowing unit propagation between those solvers. We show how such a "SAT Modulo SAT" solver naturally produces sequence interpolants as a side effect — there is no need to generate a resolution proof and post-process it to extract an interpolant. We have implemented a version of IC3 using this SAT Modulo SAT solver, which solves both more SAT instances and more UNSAT instances than PDR and IC3 on each of the 2008, 2010, and 2012 Hardware Model Checking Competition benchmarks.**

***Index Terms*—SAT, IC3, PDR, Interpolants**

I. Introduction

SAT solvers play a central role in many hardware and software model-checking techniques. In this paper, we introduce three inter-dependent contributions, culminating in an improved state-of-the-art model-checker. First, we describe a way to compose multiple SAT solvers into chains and trees, in order to efficiently solve problems that have an underlying "modular" structure (for example, instances produced by unrolling a transition function). We show that this technique can be thought of as a nested SAT Modulo Theory (SMT) solver, and that we can apply techniques from lazy SMT solvers to improve the performance of this "SAT Modulo SAT" solver. Our nested SAT solver provides a general-purpose way to take advantage of locality while solving a CNF with (known) structure.

Secondly, we show that our SAT Modulo SAT solver produces sequence interpolants [5], [21], by extending previous work by Chockler et al. [6]. These sequence interpolants are produced without requiring explicit proof-traces.

Our third contribution is to demonstrate that our SAT Modulo SAT solver can be useful in practice, by implementing a variant of IC3 [4] using it (and, implicitly, the sequence interpolants we produce).[1] We show that the resulting model checker outperforms both IC3 and PDR [11] on the 2008, 2010, and 2012 Hardware Model Checking Competition benchmarks.

II. Modular SAT Solvers

Given a *partitioned CNF* formula $\phi_0, \phi_1, \ldots \phi_n$, where each ϕ_i is a set of clauses, the *partitioned Boolean satisfiability problem* consists of determining the satisfiability of $\bigcup_{i=1}^{n} \phi_i$. Here, we will consider only cases where the partitioning into clause sets is explicitly specified or can be observed directly from the underlying problem. We will refer to the clause sets $\phi_0, \phi_1, \ldots$ as *modules*, and to any SAT solver that is designed to solve such a partitioned CNF, as a *modular SAT solver*.

Obviously, to solve a partitioned CNF one could simply merge all the partitions and solve the resulting CNF using a standard SAT algorithm, but doing so loses any structural information that might have been present in the partitioned CNF. Real-world problems often possess a high degree of modular structure (e.g., formulas derived from real-world circuits or software), so this structural information may be useful. An approach that has been investigated widely in the literature is to find the variables that are shared between modules (we will refer to these as *interface variables*), and to assign them first. Because the partitions ϕ_i are independent of each other under any complete assignment to these variables, each module can then be solved independently [17] (and in parallel [15]). Unfortunately, this method requires a potentially exponential number of assignments to the interface variables to be tested. Alternatively, the interface variables can simply be used to inform a static decision heuristic. Many strategies for partitioning a CNF have been investigated for this latter approach (e.g., [1], [9], [13]).

In this paper, we describe a new modular SAT algorithm. This algorithm relies upon three existing capabilities of typical incremental SAT solvers (such as MiniSat [10] and PicoSat [2]), namely:

1) Incremental SAT solvers allow for a CNF to be solved repeatedly as new clauses are added (maintaining heuristic values and learned clauses between runs).
2) They allow for the *temporary* addition (and subsequent removal) of unit clauses in the CNF. Equivalently, they allow for the CNF to be solved repeatedly under the temporary assumption of different partial assignments.
3) When the CNF is not satisfiable under such a partial assignment, they can return a concise clause over just the assumed unit clauses that 'explains' why those units cannot be mutually true in the CNF. This clause will include only variables that are common to both the assumed unit clauses and to the CNF being solved under the assumption.

A simple, recursive algorithm is shown in Alg. 1. To our knowledge, we are the first to propose solving a general partitioned CNF in this way; however, this algorithm is very closely related to several other approaches, as we will discuss below. Subsequently, we will build on this algorithm and arrive

[1] Just to forestall a potential point of confusion: Though we apply some techniques from lazy SMT solvers, we are not extending IC3 to handle theories other than SAT. This has been done (see, e.g., [7]), but is orthogonal to our contribution here. Our use of SMT techniques is instead to directly speed up the core Boolean satisfiability reasoning of IC3.

ISBN 978-0-9835678-3-7/13.

Algorithm 1 (Unoptimized) Modular SAT Solver

Input: Partial assignment α_i, set of clauses ϕ_i.
Output: TRUE if $\bigcup_{j=0}^{i} \phi_j$ is SAT under α_i, else
a conflict-clause which is inconsistent with α_i and contains only variables common to α_i and $\bigcup_{j=0}^{i} \phi_j$.
//The (initially empty) sets of conflict-clauses L_{ϕ_i} maintain
//the invariant $\bigcup_{j=0}^{i-1} \phi_j \Rightarrow L_{\phi_i}$.
function MODULARSOLVE(α_i, ϕ_i)
  loop
    if $\phi_i \cup L_{\phi_i} \cup \alpha_i$ is SAT **then**
      $\alpha_{i-1} \leftarrow$ satisfying assignment to $\phi_i \cup L_{\phi_i} \cup \alpha_i$
      if $i = 0$ **then**
        return TRUE
      else
        $c \leftarrow$ MODULARSOLVE(α_{i-1}, ϕ_{i-1})
        if c = TRUE **then**
          return TRUE
        else
          $L_{\phi_i} \leftarrow L_{\phi_i} \cup \{c\}$
    else
      $c \leftarrow$ conflict-clause for $\phi_i \cup L_{\phi_i} \cup \alpha_i$
      return c
  end loop

at a new, improved method: our SAT Modulo SAT solver, described in Alg. 3.

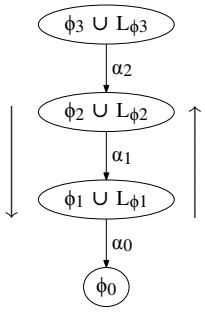
Alg. 1 operates progressively over the modules, first attempting to solve module ϕ_n, which will either be unsatisfiable, or will provide us with an assignment α_{n-1} (Figure 1). It then recursively solves ϕ_{n-1} under assignment α_{n-1}. Note that while α_{i-1} is a complete assignment to ϕ_i, it may be a partial assignment to ϕ_{n-1}.

If a module ϕ_i cannot be satisfied under α_i, the incremental SAT interface produces a learned clause c over the variables in α_i. We will refer to such a clause as an *interface clause*. We add this interface clause c into the set L_{ϕ_i}, which will be conjoined with ϕ_i when solving it in all subsequent iterations. With the addition of c, the conjunction $\phi_i \wedge L_{\phi_i}$ will now either be unsatisfiable (in which case we are done), or force the solver into a new solution with a different assignment to the interface variables.

Correctness of Alg. 1 follows from straightforward induction on i; termination is guaranteed, because L_{ϕ_i} is strengthened at every call (unless a satisfying assignment is found, in which case the algorithm terminates and returns TRUE).

One feature of Alg. 1 is that it expects an ordering over the modules. The specific ordering chosen is effectively a heuristic, and in some cases a good choice may be obvious from the problem context (*e.g.*, bounded model checking); the algorithm is correct even if the order is chosen arbitrarily (though this would impact the meaning of the interpolants produced by the algorithm, examined in the next section), and is also correct if the ordering is changed dynamically at runtime. Though we do not explore it here, one could consider randomly permuting the order, or applying a dynamic heuristic to adjust the order as the algorithm proceeds, rather

Modules are solved top to bottom, passing temporary assignments α_i down the chain from ϕ_{i+1} to ϕ_i.



When a module ϕ_i is unsatisfiable under assignment α_i, a learned interface clause c is derived from ϕ_i and added to $L_{\phi_{i+1}}$.

Fig. 1: A chain of four SAT Modules.

than relying upon a static ordering. We also observe that Alg. 1 can be trivially extended to operate over a tree-ordering of the modules: instead of MODULARSOLVE(α_i, ϕ_i) recursively calling just MODULARSOLVE(α_{i-1}, ϕ_{i-1}), it would make a recursive call for each of its children. Alg. 1 is related to several recent algorithms. For example, in the case of exactly two modules, Alg. 1 is equivalent to (the simplest case of) the proofless interpolant computing algorithm introduced in [6], and it also resembles typical all-SAT procedures [12] and some 2QBF solvers [18]. We can also think of this algorithm as forming a nested series of counter-example-guided abstraction-refinement loops: each $\phi_i \cup L_i$ is an abstraction of its conjunction with the modules below it, and the learned interface clauses returned from later modules serve to refine that abstraction by eliminating spurious counter-examples.

If we obtain the modules by unrolling a transition function (with one module per time step), then Alg. 1 is roughly equivalent to a simplified, slightly re-organized version of the recursive cube-blocking procedure at the core of IC3 and PDR; in Section III, we will examine this connection to IC3 in more detail. However, the modular SAT solver we have described here, and the proof above, is general to the case where the partitions ϕ_i are not all copies of the same transition function.

A. SAT Modulo SAT

Alg. 1 has the benefit that it can be implemented directly using the incremental interface exposed by typical SAT solvers without any modifications. Unfortunately, in practice it performs poorly, because unit propagation between modules is delayed until each previous module is completely solved, and many learned clauses must be passed up the chain to eliminate parts of the search space that would normally have been pruned by unit propagation alone (if we were solving the complete conjunction of the modules). In the worst case, this can lead to an exponential slow-down, as an exponential number of solutions from ϕ_1 might need to be produced before finding one that satisfies ϕ_2.

Lazy SAT Modulo Theory (SMT) solvers [19] face many of the same challenges as our naïve modular SAT solver above, and we can adopt the mechanisms they use to address these challenges by observing that Boolean satisfiability is itself an ideal candidate to be a theory in a lazy SMT solver. Instead of first finding a complete satisfying assignment α_i to ϕ_i

and then solving ϕ_{i-1} under it, we modify Alg. 1 to eagerly perform unit propagation on ϕ_{i-1} (and ϕ_{i-2}, etc.) as the partial assignment to ϕ_i is being constructed, returning a conflicting interface clause as soon as the partial assignment of ϕ_i would lead to a conflict in ϕ_{i-1} or a lower module.

In Alg. 2 and 3, we describe in detail the changes needed to convert a typical incremental SAT solver into an efficient modular SAT solver using this eager unit propagation across modules. In the interest of space, we will assume the reader is familiar with MiniSat [10] and describe the necessary modifications in reference to that implementation.

To apply this eager unit propagation across modules efficiently, we introduce a new method, PROPAGATEALL, which first applies unit propagation locally on the current module ϕ_i (by calling PROPAGATE) and then recursively propagates any resulting assignments to the interface variables in the next module, ϕ_{i-1}. If this leads to a conflict, then MiniSat's ANALYZEFINAL method returns a conflict clause over the interface variables.

If unit propagation of the interface assignments is successfully applied to ϕ_{i-1}, then we check if that unit propagation assigned any new literals on the interface between ϕ_i and ϕ_{i-1}. If any such assignments were made, then we again propagate those assignments locally, and continue in this way passing assignments back and forth between adjacent modules until we reach a fixed point or a conflict.

In order to accomplish this eager unit propagation efficiently, we make one additional change. When a literal is propagated, CDCL SAT solvers store a reference to the clause that was unit, so that they can explore it later during conflict analysis. If an assignment is made to the interface by ϕ_{i-1} in Alg. 2, then we would not actually have any such clause in the SAT solver for ϕ_i to use as a reason for this assignment. We can create such a clause on the interface variables by calling MiniSat's ANALYZEFINAL, but rather than call this eagerly after each unit propagation from ϕ_i, we instead set the literal's "reason" to a temporary placeholder value.

MiniSat accesses these reason references only during conflict analysis, using a helper method, REASON. We modify REASON to check for that placeholder value, and replace it with a clause produced by calling ANALYZEFINAL on ϕ_{i-1}. In this way we create the reason clauses for units propagated from ϕ_{i-1} to ϕ_i only if needed by conflict analysis (efficient SMT solvers take a similar approach).

Finally, we modify the main CDCL loop (Alg. 3) in two ways. First we alter it to call PROPAGATEALL instead of PROPAGATE. Second, once ϕ_i is entirely assigned, we modify it to recurse on ϕ_{i-1}.

Applying unit propagation eagerly allows the SAT solver for each module to prune its search space early, while the lazy reason construction reduces the number of trivial interface clauses that would otherwise have to be learned gradually and passed up through the chain of modules. Taken together, we refer to the modular SAT solver using these SMT-inspired optimizations in Algs. 2 and 3 as a "SAT Modulo SAT" solver.

Algorithm 2 PROPAGATEALL method applies intra-module and inter-module unit propagation. Note that we rely upon a list of assigned literals, $trail_{\phi_i}$, maintained for each module between calls.

```
function PROPAGATEALL(φ_i)
    loop
        // Call unit propagation on φ_i
        c ← PROPAGATE(φ_i)
        if c is a clause then
            return c    // c is a learned clause
        else if i=0 then
            return TRUE
        // Collect all new assignments to interface variables
        if trail_{φ_i} \ trail_{φ_{i-1}} = ∅ then
            return TRUE
        // Propagate new assignments in φ_{i-1}
        for all l ∈ (trail_{φ_i} \ trail_{φ_{i-1}}) do
            ENQUEUE_{i-1}(l)
        if PROPAGATEALL(φ_{i-1}) ≠ TRUE then
            c = ANALYZEFINAL(trail_{φ_i}, φ_{i-1})
            L_i ← L_i ∪ {c}
            ADDCLAUSE(c)
            return c
        else if (trail_{φ_i-1} \ trail_{φ_i}) = ∅ then
            return TRUE // No new interface assignments
        else
            // Propagate new assignments from φ_{i-1} in φ_i
            for all l ∈ (trail_{φ_i-1} \ trail_{φ_i}) do
                ENQUEUE_i(l)
                // Mark reason for lazy construction
                reasons[var] ← 'LazyPlaceholder_{i-1}'
    end loop

function REASON(var)
    if reasons[var] = 'LazyPlaceholder_{i-1}' then
        c ← ANALYZEFINAL(var,φ_{i-1})
        L_i ← L_i ∪ {c}
        ADDCLAUSE(c)
        reasons[var] ← c
    return reasons[var]
```

B. Interpolants as Side Effects

Interpolants [16] form a core part of many recent SAT-based model checkers, including IC3. Normally, interpolants are constructed by analyzing a resolution proof-trace, which must be generated by a SAT solver as it is solving an instance. This introduces an overhead into the solving process (for this reason, recent work ([6], [20]) has investigated alternative methods that do not require constructing an (explicit) proof trace).

We now show that the sets of learned interface clauses L_{ϕ_i} collected between each module in Alg. 1 form valid interpolants. Taken together, these successive interpolants form a sequence interpolant [21]. An alternative proof for the case of exactly two modules can be found in [6]. For simplicity, we

Algorithm 3 The main CDCL loop of our SAT Modulo SAT solver, using the PROPAGATEALL method. We integrate the recursive call to the next solver directly into the search loop. Other than the changes here and in the PROPAGATEALL method, our implementation follows MiniSat 2.2 [10]. Alg. 3 is a direct replacement for Alg. 1.

```
function ModularSolve(α_i, φ_i)
    loop
        conflict ← PropagateAll(φ_i)
        if conflict is a clause then
            if DecisionLevel( ) = 0 then
                return conflict
            c ←Analyze(conflict)
            Backtrack( )
            addClause(c)
        else
            if exists an unassigned lit ∈ α_i then
                l ← lit
            else
                l ← pickBranchLit( )
            if l = NULL then
                // trail_φi is a satisfying assignment to φ_i
                if i = 0 then
                    c ← True
                else
                    c ← ModularSolve(trail_φi, φ_{i−1})
                if c = True then
                    return True
                else
                    // Learn clause c from φ_{i−1}
                    L_i ← L_i ∪ {c}
                    Backtrack( )
                    addClause(c)
            else
                newDecisionLevel( )
                Enqueue_i(l)
    end loop
```

describe our proof in terms of the unmodified Alg. 1, but it holds equally well for the optimized SAT Modulo SAT solver.

Given a CNF partitioned into two parts, ϕ_A and ϕ_B, with $\phi_A \cup \phi_B$ unsatisfiable, an interpolant between ϕ_A and ϕ_B is any set of constraints I with the following three properties:

1) ϕ_A implies I.
2) $I \cup \phi_B$ (i.e., the conjunction of the constraints) is unsatisfiable.
3) I contains only variables common to ϕ_A and ϕ_B.

First, consider Alg. 1 with only two modules. On an unsatisfiable instance, Alg. 1 terminates only when the top-most module ϕ_1, combined with the interface clauses L_{ϕ_1} it has learned from module ϕ_0, does not have any satisfying solutions. So at termination (on an unsatisfiable instance), $\phi_1 \cup L_{\phi_1}$ must be unsatisfiable. We also have that $\phi_0 \Rightarrow L_{\phi_1}$, because L_{ϕ_1} consists only of clauses implied by ϕ_0. Finally, the incremental SAT solver interface guarantees that each clause in L_{ϕ_1} contains only variables that are common to ϕ_1 and ϕ_0. These three conditions together satisfy the definition of an interpolant between ϕ_1 and ϕ_0.

Next, consider an unsatisfiable chain of three modules, ϕ_2, ϕ_1, and ϕ_0. There are *two* interpolants that are constructed by Alg. 1: An interpolant L_{ϕ_2} between ϕ_2 and $(\phi_1 \wedge \phi_0)$, and an interpolant L_{ϕ_1} between $(\phi_2 \wedge \phi_1)$ and ϕ_0.

In this three module chain, the argument that L_{ϕ_1} forms an interpolant is the same as above. The argument that L_{ϕ_2} forms an interpolant is similar, except that the clauses collected in L_{ϕ_2} are implied by the conjunction $\phi_1 \wedge \phi_0$, rather than by ϕ_0 alone. This is the case even though in Alg. 1 module ϕ_2 is only ever passed interface clauses constructed by ϕ_1 (and never by ϕ_0), because module ϕ_1 may itself have been passed interface clauses from module ϕ_0, and may then have derived new constraints based on those facts that are subsequently passed to module ϕ_2.

In general, at termination on an unsatisfiable instance, it must either be the case that $\phi_n \wedge \phi_{n-1} \wedge \ldots \wedge \phi_i$ is by itself already unsatisfiable (in which case L_{ϕ_i} is the empty set, and a trivial interpolant), or that $\phi_n \wedge \phi_{n-1} \wedge \ldots \wedge \phi_i \wedge L_{\phi_i}$ is unsatisfiable, in which case L_{ϕ_i} is a valid interpolant between the conjunctions $\phi_n \wedge \ldots \wedge \phi_i$ and $\phi_{i-1} \wedge \ldots \wedge \phi_0$.

III. IC3 Using SAT Modulo SAT

The modular SAT solver we have described here operates on an ordered sequence of CNF modules; a natural use case would be to apply it to bounded model checking [3] by constructing one module per time step, and incrementally adding new modules as time steps are added. Unfortunately, performance is roughly competitive with, but not better than, an (unsophisticated) incremental bounded model checker. However, simple bounded model checking does not take advantage of the sequence interpolants that our solver naturally produces.

Sequence interpolants are not typically generated by themselves as an end goal. Instead, the primary place that sequence interpolants are used is as a component of model checking algorithms (*e.g.*, [5], [21]), most prominently in the current state-of-the-art SAT-based unbounded model checker, IC3 [4]. In IC3, sequence interpolants are created implicitly, through an incremental refinement process that is closely related to the unoptimized modular SAT solver from Alg. 1.

We now demonstrate that the SAT Modulo SAT solver we presented above is useful in practice by creating a version of IC3 based on it and the sequence interpolants it produces.

Our implementation closely follows the PDR [11] variant of IC3, which we do not have space to recount in full. We will assume the reader is familiar with PDR, and describe only our changes here. Modifying PDR's algorithms to use the modular SAT solver will entail some non-trivial changes, which we describe below. As well, while building our solver, we developed some minor improvements to the general IC3 algorithm; we will show below that these minor changes are indeed improvements, but that the most important performance improvement is due to our SAT Modulo SAT solver.

Algorithm 4 The cube-blocking procedure for the stack-based variant of IC3, using a modular SAT solver. Notice that the stack is actually completely eliminated; recursively blocking the cube is directly handled by the modular SAT solver (MODULARSOLVE calls either Alg. 1 or Alg. 3 above). In contrast to IC3, all the newly generated blocking clauses are collected and generalized at the end.

```
function MODULARBLOCKCUBE(TCube s0)
    i ← s0.frame−1
    if MODULARSOLVE(s0.cube, φ_i) then
        return FALSE // Counter-example found
    else
        COLLECTALLCLAUSES(i)
        return TRUE

function COLLECTALLCLAUSES(t)
    // Collect new interface clauses from the first t solvers
    // We assume these are stored in vectors
    // newInterfaceClauses_i for each frame
    for i ← 1 . . . t − 1 do
        for all Clause c ∈ newInterfaceClauses_i do
            MARKSOLVER(i) // Needs clause propagation
            c ← GENERALIZE(c)
            // Attempt to propagate c forward until it fails
            j ← EAGERPROPAGATECLAUSE(c, i)
            F[j].ADD(c)
        newInterfaceClauses_i ← ∅

function EAGERPROPAGATECLAUSE(Clause c, from)
    // Propagate clause c forward as far as we can
    for i ← from . . . F.size()−1 do
        if not PROPAGATECLAUSE(c, i) then
            return i
        return i
```

The central part of IC3 is the cube-blocking procedure (in PDR, "RECBLOCKCUBE"). There are two major variants of this procedure. The simpler, 'stack-based' version takes an assignment to the flops (a cube) that is known to lead to the negated property, and incrementally strengthens the interpolants between each time frame until they are sufficient to block that cube in the last time frame. In Alg. 4, we show how we can use a modular SAT solver (either Alg. 1 or Alg.3) to replace RECBLOCKCUBE . Intuitively, cube-blocking in the stack-based variant of IC3 is performing *almost* the same function as the simple recursive modular SAT solver of Alg. 1, with a few extra steps added. By re-arranging this code to separate out the part that closely matches Alg. 1 we then make it possible to replace it with the more complicated SAT Modulo SAT solver in Alg. 3 as well.

The match is not exact. The most obvious difference is that IC3 applies inductive generalization [4] to drop literals from conflict clauses as they are added to the interpolants. Unfortunately, the solvers for each time step are maintaining state between calls in the modular SAT solver, which would be overwritten during inductive generalization. One way to resolve this would be to keep an extra SAT solver, not part of the SAT modulo SAT solver, and use that to apply inductive generalization as conflict clauses are learned. This would allow us to apply inductive generalization at the same point as IC3, at the cost of extra memory usage. A second option, which we take in Alg. 4, is to delay inductive generalization until after the complete call to the modular SAT solver (during which many interface clauses may have been learned), and then subsequently apply inductive generalization to each new clause. This allows us to re-use the solvers from our modular SAT solver for generalization.[2]

Algorithm 5 The cube-blocking procedure for the priority-queue based version of IC3 using a modular SAT solver. This function is a replacement for the RECBLOCKCUBE procedure of PDR. We show here the *keepCubes* option discussed below. With *keepCubes* set, we keep the last time frame's TCubes in the priority queue for the next iteration rather than discarding them (as PDR does).

```
function MODULARBLOCKCUBEPRIORITY(TCube s0)
    Q.ADD(s0)
    while Q.SIZE( ) > 0 &&
          Q.PEEK( ).frame < F.SIZE( ) do
        TCube s ← Q.POPMIN( )
        if not ISBLOCKED(s) then
            if not MODULARBLOCKCUBE(s) then
                return FALSE // Counter-example found
            else
                Q.ADD(COLLECTALLCUBES(s.frame))
                if keepCubes || s.frame < F.SIZE() -1 then
                    Q.ADD(TCube(s.cube, s.frame + 1))
        else if keepCubes && s.frame < F.SIZE() -1 then
            Q.ADD(TCube(s.cube, s.frame + 1))
    return TRUE

function COLLECTALLCUBES( t)
    // Collect all satisfying assignments to the flops
    // found during MODULARSOLVE. We assume these
    // were stored for frame i in vector flopAssignments_i.
    for i ← 1 . . . t − 1 do
        for each assignment a ∈ flopAssignments_i do
            Q.ADD(TCube(a, i + 1))
        flopAssignments_i ← ∅
```

Another difference is that one of the original selling points of IC3 was that it does not require the transition function to be unrolled; instead, a growing set of sequence interpolants (with some special properties discussed below) are maintained by re-using a single transition function between consecutive interpolants in the sequence. By instantiating a separate copy

[2]Another subtlety is that when we apply inductive generalization to a clause from module ϕ_i, we re-use the SAT solver for ϕ_i from our modular solver, but call its normal, non-modular SOLVE method (which does not recursively solve the other modules in the chain). An alternative option would be to use the entire modular SAT solver chain during generalization, which would increase the chance of dropping literals from the conflict clauses, but at the cost of introducing an additional linear time factor (in the number of modules) into generalization.

of the transition function for each module ϕ_i in our modular SAT solver, we are giving up this near-constant memory usage. However, recent versions of PDR have made the same time-space trade-off, to avoid the cost of tracking which learned clauses correspond to which time frame.

A more substantial difference between our SAT modulo SAT solver and IC3 is that IC3 requires the interpolants for each time frame in the sequence to be constructed from a subset of the clauses that make up the interpolant for the previous time frame. We cannot combine the trick IC3 usually applies for this with Alg. 3, and must instead add a non-deterministic self-loop to the transition function (by adding an extra input to the circuit that, when true, forces the flops to their reset state). This extra non-determinism might be expected to slow down the SAT solver.[3] However, because our solver (like IC3) always solves its time frames in reverse order, the self-loop will always be disabled by simple unit propagation before any decisions must be made in a given time frame. This makes such a self-loop in the transition function almost cost-free.

Having made these changes, we can directly use a modular SAT solver (either the simple recursive Alg. 1 or the more complex SAT Modulo SAT solver Alg.3) to implement the stack-based cube-blocking procedure from IC3 (see Alg. 4).

Efficient versions of IC3, including PDR, maintain a priority queue of cubes to block rather than a stack. In this variant, when IC3 blocks a cube, it generates a new cube with the same flop assignment, but at the next time frame. This allows IC3 to discover counter-example traces that are longer than the number of time frames currently being examined [4], while at the same time improving the overall performance of IC3 [11]. In order to support this, we need to make our implementation slightly more complicated (see Alg. 5), as well as change the modular SAT solver slightly, so that it records each complete satisfying assignment of the flops in each time frame. This is a trivial one line change to the SAT modulo SAT solver. In Alg. 5, we assume that the flop assignments found for time frame i have been stored in the vector $\mathit{flopAssignments}_i$.

The priority queue version of IC3 then proceeds by repeatedly popping the lowest TCube s off the queue (a TCube is a tuple of a cube and the time frame it corresponds to), solving ϕ_0 under $\phi_1 \ldots$ under $\phi_{s.frame}$ under $s.cube$, and then adding all the cubes that were found during that process into the queue (see CollectAllCubes). Effectively, this results in a combination of the priority-queue with the modular SAT solver's natural stack-based order for exploring cubes. As we will show below, this re-ordering appears to have a negative impact on performance, but one that is more than made up for by the use of the modular SAT solver.

With these changes, and otherwise following PDR's implementation (including applying ternary simulation, which we apply to α_i just before the loops in Alg. 1 and 3), we used our modular solver to implement a competitive version of the PDR variant of IC3. As we will show below, in addition to solving as many or more instances as either PDR or IC3 on three major benchmark sets, this procedure solves many *different* instances that were not previously solved by either PDR or IC3.

A. Additional Changes to IC3

We also introduce two additional alterations to IC3 to further improve our solver. The first change is fairly minor. In the priority queue variant of IC3, when a cube is blocked at time frame i, it is re-enqueued at frame $i+1$. However, if i is the last currently expanded time frame, the cube is simply discarded. Instead, we keep these cubes and enqueue them into the priority queue at frame $i+1$, and keep them in the queue for the next iteration (at which point time frame $i+1$ will have been explored). This is shown in Alg. 5, when the *keepCubes* flag is set. We only ever discard cubes from the last time frame if they are syntactically blocked. We have also explored keeping all such clauses even if they are blocked syntactically in the last time frame, and it seems to lead to only a slight decrease in performance to do so.

A more significant change addresses a drawback of IC3 (including the PDR variant). IC3 always attempts to propagate clauses from the first to the last time frame at each iteration. As a result, IC3 requires at least quadratic time in the number of frames, and that by itself can lead to unacceptable slow-downs on instances that require many iterations to be explored, even if the instance is otherwise trivial. Just such an example has been encountered in practice by users of the Z3 [8] implementation of PDR.[4]

We observe that it is not typically necessary to try propagating clauses all the way from the lowest time frame at each iteration. Instead, we have found it sufficient in practice to propagate only from the lowest *strengthened* time frame to the last, at each iteration (see Alg. 6). This is a very simple change that improves performance when an instance is explored to a very deep time bound. Informal testing on Z3's PDR variant [14] has shown that this change improves performance on the example referenced earlier.

In principle, failing to propagate clauses from the first time frame may lead to a loss of IC3's convergence guarantees. If this were a concern, it would be sufficient to force clause propagation to periodically start from the first time frame — something we have tried and found not to lead to substantial performance improvements in practice.

[3] IC3 enforces this property by ensuring that all clauses in each interpolant hold at the reset state. In cases where it would learn such a clause that does not hold at the reset state, it weakens the clause by appending a literal from the reset state that does not already appear in the clause. Such a literal is guaranteed to exist, because if no literals in the cube were opposite the polarity of the reset state, then IC3 would have found a counter-example (and exited). That literal can be used to weaken the conflict clause so that it is satisfiable at the reset state, while still blocking the cube.

We cannot combine this trick with unit propagation across modules as in Alg. 3, because such propagation may occur when an arbitrary partial assignment has been made to the flops. This partial assignment might not contain any literals opposite the reset state, in which case we would not be able to weaken the clause as IC3 does while still blocking the assignment.

[4] See, *e.g.*, http://stackoverflow.com/q/15946304

Algorithm 6 Faster clause propagation, by not attempting to propagate clauses from time frames that did not require strengthening in the current iteration.

```
function PROPAGATECLAUSES
    lowest ← 0
    for i ← (F.size()-1) ... 0 do
        if not SolverIsMarked(i) then
            lowest← i + 1
            break
        clearSolverMark(i)
    for k ← lowest ... F.size()−1 do
        for all clauses c ∈ F[k] do
            if PROPAGATECLAUSE(c,k + 1) then
                F[k].remove(c)
                F[k + 1].add(c)
        if F[k].size()=0 then
            return 'Invariant Found'
```

IV. Experimental Results

Our implementations of both Alg. 1 and the SAT Modulo SAT solver described in Section II-A are based on MiniSat 2.2 [10], a prominent incremental SAT solver that has served as a basis for many successful SAT solvers. We implemented IC3 using this solver as described above in Alg. 5.[5]

We compare to both the publicly released IC3, and also to the current implementation of PDR in ABC (version 1.01). This implementation of PDR is also part of the SUPERPROVE model checker that won the Hardware Model Checking Competition in 2010, 2011, and 2012.

Experiments on the 2008 instances were conducted on 32-bit 3.2GHz Intel Xeon machines with 2 MB cache under openSUSE 11.1, using 15 minute timeouts and 1500 MB memory limits. Experiments for the 2010 and 2012 instances were conducted on 64-bit, 6-core, 2.6GHz Intel Xeon machines with 12 MB cache running Red Hat Linux 5.5, using 15 minute timeouts and 7000 MB memory limits. These conditions closely match those of the 2008 and 2012 competitions, respectively. When testing each model checker (including PDR and IC3), we first used ABC to apply DAG-aware rewriting for preprocessing the circuit (using the 'rewrite' command).[6]

Using our PDR implementation with the SAT Modulo SAT solver, but without the last two improvements to IC3 from Section III-A, performance is comparable to both IC3 and PDR (see the column, 'SMS', of Table I). If we substitute the unoptimized Alg. 1 for the SAT Modulo SAT solver, performance drops substantially on all benchmarks (see column 'No SMS'). This gives us confidence that our SAT Modulo SAT solver is indeed a major improvement to Alg. 1, at least in this context.[7] This model checker is also clearly competitive with IC3 and PDR — especially on the 2010 instances. Moreover, on closer inspection, we also observed that this version of our model checker solved 17 *new* instances that were solved by neither IC3 nor PDR from the 2012 benchmarks, and 13 and 9, respectively, from the 2010 and 2008 sets.

We can then ask whether we can improve our solver to solve more of the instances that IC3 and PDR solve, without giving up these new instances. We accomplish exactly this, by adding the last two improvements discussed in Section III-A. These improvements allow us to solve several additional instances (all but 3 of which IC3 or PDR could already solve), without giving up *any* of the newly solved instances of our initial implementation (see column 'SMS-PDR' in Tables I and II).

From Table I, we note that on the 2008 instances, our final model checker solves just one instance fewer than the corresponding virtual best solver — the virtual best solver counts any instance solved by any solver running in that competition — under roughly the same conditions. In Table II, we split the results for each model checker into SAT and UNSAT instances, to show that for all three competitions, we always solve more SAT *and* more UNSAT instances than both IC3 and PDR. In contrast, notice that while PDR improved hugely upon IC3's performance on the 2012 instances, it actually performed slightly worse on the 2008 instances.

The improvements introduced in Section III-A, as well as the use of our SAT Modulo SAT solver, both increased

	No SMS	SMS	**SMS-PDR**	PDR	IC3	VBS
HWMCC'08	504	587	**596**	581	586	597
HWMCC'10	684	742	**749**	733	712	781
HWMCC'12	69	84	**92**	84	48	233

TABLE I: Total instances solved within 900 seconds from the 2008, 2010, and 2012 Hardware Model Checking Competitions (single property track). Including all improvements, our final implementation ('SMS-PDR') beats both IC3 and PDR on each benchmark. Notice how for the 2008 benchmarks we solve just 1 fewer instance than the virtual best solver ('VBS' — the virtual best solver counts all instances solved by *any* solver in the competition).

	SMS-PDR		PDR		IC3	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
HWMCC'08	**245**	**351**	242	339	240	346
HWMCC'10	**322**	**427**	317	416	308	404
HWMCC'12	**25**	**67**	21	63	14	34

TABLE II: Breakdown of SAT and UNSAT instances from Table I. SMS-PDR solves more SAT and more UNSAT instances than both PDR and IC3 on all three benchmarks.

[5]We have made the source code for the modular SAT solver available online, at www.cs.ubc.ca/labs/isd/Projects/ModularSAT.

[6]Results for each competition's virtual best solver is as reported in the respective competitions.

[7]At the same time, we can ask why it is the case that when using Alg. 1 instead of Alg. 3, our performance is so much worse than IC3's. As discussed in Sec. III, there are effectively just a few differences between PDR's RECBLOCKCUBE and Alg. 5, if the unoptimized modular SAT solver of Alg. 1 is used and if our last two changes to IC3 are not implemented. The performance drop *relative to IC3* in this case is likely either due to our delaying inductive generalization until later in the process, or a consequence of using a self-loop in the transition function (though we argue why this is not likely in Sec. III).

the total number of solved instances. However, the SAT modulo SAT solver by itself contributes most of the *newly* solved instances — that is, instances that we solved, but that neither IC3 nor PDR could solve. Our model checker using just the SAT modulo SAT solver solved 9, 13, and 17 *new* instances in the 2008, 1010, and 2012 benchmarks, while combining the SAT Modulo SAT solver with the changes from Section III-A solved 9, 15, and 21 such instances. On this basis, we argue that the SAT Modulo SAT solver is critical to the overall performance improvement achieved by our final model checker.[8]

We can also look at the respective memory usage of our solvers. As we remarked earlier, like current versions of ABC's PDR, we instantiate a solver for each time step, which results in roughly linear memory usage in the number of time steps. This gives up one of the original advantages of IC3, which is that it expands only one time frame at a time, which requires roughly constant memory. Both of these bounds ignore the theoretically exponential memory of the learned clauses and interpolants.

We found that our solver ran out of memory on 13, 1, and 7 instances for the 2008, 2010, and 2012 benchmarks (recall that the 2008 competition was limited to just 1.5 GB, vs 7 GB for the others). In contrast, IC3 ran out of memory on just two instances in our experiments, both from the 2012 benchmarks. However, there was only one case in which our solver ran out of memory on an instance that IC3 was able to solve - and that particular instance, from the 2008 benchmark set, was one that our solver *was* able to solve in the 2010 benchmark set (which had a higher memory limit). So, as we would expect, IC3's near-constant memory usage is an advantage on some instances.

V. Conclusion

We have introduced a novel approach for modular SAT solving, which naturally computes sequence interpolants without proofs. We have made this efficient through the use of standard techniques borrowed from lazy SMT solvers, and we have shown that this can form the basis of an efficient model checker. We have also introduced additional improvements to IC3 that should generalize to other implementations, including PDR, whether or not they utilize our SAT Modulo SAT solver. The resulting state-of-the-art model checker performs better than both PDR and IC3, for both SAT and UNSAT instances, on three competitive sets of benchmarks.

VI. Acknowledgments

We thank Armin Biere for his insights about the connection between other CNF partitioning solvers and modular solvers. We thank Nikolaj Bjorner for pointing us to the loop example cited in Section III, and for testing our faster clause propagation in Z3. We also thank the anonymous reviewers of this paper, as well as of a previous manuscript which led to this work.

This research has been supported by the use of computing resources provided by WestGrid and Compute/Calcul Canada, and by funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[8] We can also ask how the changes from Section III-A fare on their own. Implementing them in the non-SMS version of our solver does not improve performance at all, while implementing them in ABC's PDR led to 2 and 3 additional instances solved on the 2008 and 2010 benchmarks, and 3 fewer solved on the 2012 benchmarks.

References

[1] F. Aloul, I. Markov, and K. Sakallah, "MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation," *Journal of Universal Computer Science*, vol. 10, no. 12, pp. 1562–1596, 2004.

[2] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, no. 2-4, pp. 75–97, 2008.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*. Springer, 1999.

[4] A. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2011, pp. 70–87.

[5] G. Cabodi, S. Nocco, and S. Quer, "Interpolation sequences revisited," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[6] H. Chockler, A. Ivrii, and A. Matsliah, "Computing interpolants without proofs," in *Proceedings of the Eighth Haifa Verification Conference*, 2012.

[7] A. Cimatti and A. Griggio, "Software model checking via IC3," in *Computer Aided Verification*. Springer, 2012, pp. 277–293.

[8] 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.

[9] V. Durairaj and P. Kalla, "Guiding CNF-SAT search via efficient constraint partitioning," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, 2004, pp. 498–501.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 333–336.

[11] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design (FMCAD), 2011*. IEEE, 2011, pp. 125–134.

[12] O. Grumberg, A. Schuster, and A. Yadgar, "Memory efficient all-solutions SAT solver and its application for reachability analysis," in *Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 275–289.

[13] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-based decision heuristics for image computation using SAT and BDDs," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2001, pp. 286–292.

[14] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing–SAT 2012*. Springer, 2012, pp. 157–171.

[15] A. Hyvärinen, T. Junttila, and I. Niemelä, "Partitioning SAT instances for distributed solving," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2010, pp. 372–386.

[16] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*. Springer, 2003, pp. 1–13.

[17] T. Park and A. Van Gelder, "Partitioning methods for satisfiability testing on large formulas," *Automated Deduction âĂŤ CADE-13*, pp. 748–762, 1996.

[18] D. Ranjan, D. Tang, and S. Malik, "A comparative study of 2QBF algorithms," in *The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004, pp. 292–305.

[19] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 3, pp. 141–224, 2007.

[20] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in CNF," in *Computer Aided Verification*, 2013, pp. 330 – 346.

[21] Y. Vizel and O. Grumberg, "Interpolation-sequence based model checking," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. IEEE, 2009, pp. 1–8.

Better Generalization in IC3

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Email: zyad.hassan@colorado.edu, bradleya@colorado.edu, fabio@colorado.edu

***Abstract*—An improved clause generalization procedure for IC3 is presented. Whereas standard generalization extracts a relatively inductive clause from a single state, called a counterexample to induction (CTI), the new procedure also extracts such clauses from other states, called counterexamples to generalization (CTG), that interfere with the primary generalization attempt. The motivation is to enable IC3 to explore states farther from the error states than are CTIs while remaining property-focused. CTGs are strong candidates for being farther but still backward reachable. Significant reductions in the maximum depth reached by IC3's priority queue-directed explicit backward search indicate that this intention is achieved in practice. The effectiveness of the new procedure is established in two independent implementations of IC3, which demonstrate an increase of 17 and 27, respectively, in the number of solved HWMCC benchmarks.**

I. Introduction

IC3 [1], [2] is an incremental, inductive model checking algorithm for invariance properties. It operates in a demand-driven manner, generating relatively inductive lemmas in response to states that interfere with the inductiveness of the property. Lemma generation proceeds incrementally until an inductive strengthening is discovered or the lemmas guide the backward search to a counterexample trace. IC3 is SAT-based but, in contrast to other SAT-based approaches, poses relatively easy but numerous SAT queries that arise from considering single steps of a transition relation. This style of using a SAT solver keeps its memory footprint small.

One of the key components of IC3 is inductive generalization. While IC3 has an element of explicit state model checking in that it examines individual states, called counterexamples to induction (CTIs), inductive generalization makes it symbolic, allowing it to handle huge state spaces. IC3's success on a model thus hinges on its ability to generalize facts that it discovers from considering specific states.

The effectiveness of generalization depends on the connectivity of a model's state graph and its encoding. Some encodings and some models, independent of encoding, coupled with the overapproximate nature of inductive generalization, require IC3 to examine more individual states. Consider the state graph in Figure 1, where 000 is the initial state and 001 is the bad state. This model has two counterexamples to the inductiveness of the property: 110 and 100, two good states with a bad successor.

Suppose state 100 is the first CTI that IC3 finds. Since this state does not have predecessors, its negation is inductive, so that IC3 concludes it is unreachable. The unreachability of this state is a specific fact that IC3 next tries to generalize in order to prove that other states are unreachable as well. It does so by attempting to drop as many literals as possible. However, in this case no literals can be dropped. For example, if IC3 attempts to drop the third literal, the negation of the resulting cube $10-$, where $-$ indicates a don't care, is not inductive because of the predecessor 011 to state 101. If there is a cube whose negation is inductive and excludes both 100 and 101, that cube must also include 101's predecessor, 011. However, the smallest cube that includes all three states is $---$, which includes the initial state and whose negation is therefore not inductive. Similar reasoning shows that IC3 also cannot drop the first and second literals. Thus the strongest clause that can be derived through generalization only blocks the CTI itself. IC3 then has to prove that the other CTI (110) is unreachable without having learned much from the first CTI.

A state that hinders a generalization attempt (011 in the example) is called a *counterexample to generalization* (CTG): it prevents dropping a literal (the third in the example), i.e., generalizing to a larger cube. Despite being itself unreachable, state 011 causes the inclusion of an initial state into the cube that covers both it and $10-$, which in turn causes generalization to fail. In this case, it is useful to focus some effort on the CTG rather than only on the CTI. Since the negation of the CTG is inductive, IC3 can block it. Then, with its predecessor blocked, dropping the third literal of 101 succeeds. Indeed, the second literal can be dropped as well, as all predecessors of the cube $1--$ are blocked. This further expansion takes care of state 110 as well, ending the analysis.

This example motivates the improved generalization procedure described in this paper. The proposed procedure addresses CTGs that appear during the generalization of some CTI-derived relatively inductive clause. CTGs are often deep back-

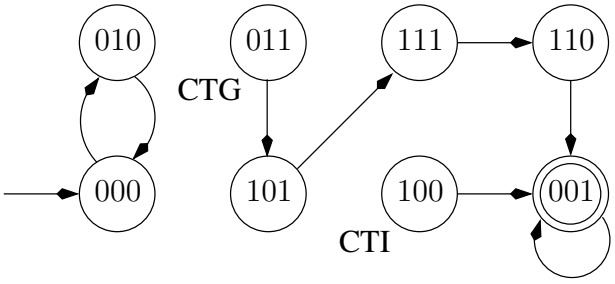


Fig. 1. Failure to generalize a clause.

ISBN 978-0-9835678-3-7/13.

ward reachable states. Addressing them reduces the depth of the explicit backward search IC3 performs and allows stronger inductive generalizations.

The proposed generalization procedure is evaluated within the implementations of IC3 in the model checkers IImc [3] and ABC [4]. Both show considerable improvement on Hardware Model Checking Competition (HWMCC) benchmarks [5].

Preliminaries are in Section II. Section III describes the proposed generalization procedure. Section IV presents the results of improved generalization on the HWMCC 2010–2012 benchmark suites. The behavior of IC3 with the improved procedure is studied in detail in Section V. Section VI discusses related work. Finally, conclusions are in Section VII.

II. Preliminaries

A. Transition Systems and Induction

Following standard practice, a *finite-state system* is represented as a tuple $S : (\bar{i},\ \overline{x},\ I(\overline{x}),\ T(\bar{i},\overline{x},\overline{x}'))$ consisting of primary inputs $\bar{i}$, state variables $\overline{x}$, a propositional formula $I(\overline{x})$ describing the initial configurations of the system, and a propositional formula $T(\bar{i},\overline{x},\overline{x}')$ describing the transition relation. Primed state variables $\overline{x}'$ represent the next state.

A state of the system is an assignment of Boolean values to all variables $\overline{x}$ and is described by a *cube* over $\overline{x}$, which is a conjunction of literals, each *literal* a variable or its negation. An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$, then s is called an F*-state*. A formula F *implies* another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G. The (in)validity of $F \Rightarrow G$ is established by querying a SAT solver for the unsatisfiability of $F \wedge \neg G$.

A *clause* is a disjunction of literals. A subclause $d \subseteq c$ is a clause d whose literals are a subset of c's literals.

A *run* of S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that $s_0 \models I$ and for each adjacent pair (s_j, s_{j+1}) in the sequence, $\exists\bar{i}.(\bar{i}, s_j, s'_{j+1}) \models T$. That is, a run is the sequence of assignments in an execution of the transition system. A state that appears in some run of the system is *reachable*.

Checking a *safety property* of S is reducible to checking an invariance property [6]. An *invariance property* $P(\overline{x})$, a propositional formula, asserts that only P-states are reachable. P is *invariant* for the system S (that is, S-invariant) if indeed only P-states are reachable. If P is not invariant, then there exists a finite *counterexample* run $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. An invariance property $P(\overline{x})$ is *inductive* if

1) (*initiation*) every initial state satisfies the property: $I(\overline{x}) \Rightarrow P(\overline{x})$; and
2) (*consecution*) every transition from a P-state leads to a P-state: $P(\overline{x}) \wedge T(\bar{i},\overline{x},\overline{x}') \Rightarrow P(\overline{x}')$.

While an inductive property P is invariant, the converse is not necessarily true. In this case, it is customary to seek an *inductive strengthening* of P, which is a formula F such that $F \wedge P$ is inductive.

An assertion F is *inductive relative to* another assertion G, possibly containing primed variables, if

1) every initial state satisfies F: $I(\overline{x}) \Rightarrow F(\overline{x})$; and
2) F satisfies consecution under assumption G: $G(\overline{x},\overline{x}') \wedge F(\overline{x}) \wedge T(\bar{i},\overline{x},\overline{x}') \Rightarrow F(\overline{x}')$.

B. An Overview of IC3

IC3 maintains a sequence of overapproximations F_i to sets of states reachable within i steps, for $0 \leq i \leq k$, where F_k is the frontier. Each F_i is a conjunction of the property P with an initially empty set of clauses. For each $k > 0$, IC3 refines the F_i's for $i \leq k$ as needed to prove inductiveness of P relative to F_k. This refinement is property-driven: a counterexample to the inductiveness (CTI) of the property, which is an F_k-state with a $\neg P$-successor, triggers IC3 to derive a clause to block it. If successful, it applies induction to generalize the clause to block many more states than the CTI alone. It then adds the generalized clause to F_i for all $i \leq k$.

Otherwise, it explores (transitive) predecessors of the CTI to derive supporting strengthening clauses until the original CTI can itself be addressed relative to F_k. This exploration of concrete predecessors is guided by a priority queue of pairs of states and frame indices: (s, i) represents the obligation that state s must be inductively excluded relative to F_i, i.e., proved unreachable for at least $i + 1$ steps. Obligations are handled in lowest-index-first order, guaranteeing termination. IC3 aggressively generalizes from states: once it addresses (s, i) by finding a clause $c \subseteq \neg s$ that is inductive relative to some F_j, $j \geq i$, IC3 adds obligation $(s, j+1)$ to the queue if $j < k$. This aggressive strategy not only facilitates early discovery of mutually inductive clauses; it also allows IC3 to find deep counterexamples even when k is small.

When no CTIs remain (for F_k), IC3 checks each clause of each F_i to determine if it can be propagated forward, i.e., if it has become inductive relative to F_i since its creation because of subsequent strengthening of F_i. In the process, IC3 determines whether any F_i has become an inductive strengthening of the property, in which case the property is declared to hold. If not, it increments k and "bootstraps" the new frontier F_k with all clauses that are inductive relative to F_{k-1}. This process continues until IC3 finds an inductive strengthening of the property or finds a counterexample by following a sequence of CTIs back to an initial state.

IC3 generalizes a clause by using induction to guide the dropping of literals. IC3's generalization procedure is described in Listing 1. Notice, when reading the pseudocode, that cubes are passed by reference. The minimum-inductive clause procedure (MIC) attempts to drop each literal in turn from q, calling down to validate each potential strengthening of the clause (and, as a side effect, to further strengthen the clause). If down reports that the literal cannot be dropped, MIC returns it to the clause.

Given a cube q, the down procedure seeks the maximal inductive subclause of $\neg q$. It returns true if found and false if no inductive subclause exists. The down procedure effectively computes an overapproximation of states backward reachable

Listing 1. IC3 generalization procedure.

```
void MIC(q: cube ref, i: level):                          1
  foreach literal l in q:                                 2
    q̂ := q \ l                                            3
    if down(q̂, i):                                        4
      q := q̂                                              5
                                                          6
bool down(q: cube ref, i: level):                         7
  while true                                              8
    if I ⇏ ¬q:                                            9
      return false                                        10
    if F_i ∧ ¬q ∧ T ⇒ ¬q′:                                 11
      return true                                         12
    with (F_i ∧ ¬q)−state s:                              13
      q := q ⊔ s                                          14
```

from a given state set. It limits the cost by maintaining sets of states in the form of a single cube. If upon reaching a fixpoint, the cube does not include any of the initial states, the cube represents a set of states that are unreachable in $i+1$ steps. Denoting the cube at iteration j by q_j, each fixpoint iteration queries the SAT solver for the existence of an $(F_i \wedge \neg q_j)$-predecessor to some q_j-state. The absence of such a predecessor indicates the inductiveness of $\neg q_j$ relative to F_i. Otherwise, there is an $(F_i \wedge \neg q_j)$-state s that is a predecessor to some q_j-state. A new cube q_{j+1} is formed by taking the common literals in q_j and s (denoted by $q_j \sqcup s$). The number of literals in the cube thus strictly decreases in every iteration, effectively expanding the set of states in q_j.

The MIC procedure can be optimized using the up procedure [7], which is outside the scope of this paper.

III. Addressing Counterexamples-to-Generalization

A. Presentation of the Procedure

Keeping only the common literals of q_j and s provides an overapproximating union over state sets—a *join* in the cube lattice. While this operation responds to the need to include the q_j-predecessor s in the state set described by q_{j+1}, it also typically brings in other F_i-states. Therefore, even when all q_0-states are unreachable, down (eventually) fails if, through overapproximation, it incorporates a reachable state.

State s is called a *counterexample to generalization* (CTG) since it is encountered in the context of dropping a literal (in MIC) in order to generalize a cube. Unlike CTIs, states brought in as a result of dropping a literal or joining are not necessarily backward reachable from the error. On one hand, if s is backward reachable—and it represents a set of deep backward reachable states—then addressing it could save IC3 from having to explicitly traverse the state graph from the error state to s. On the other hand, if s is neither backward nor forward reachable, it could still obstruct generalization: when it is joined with q_j to form q_{j+1}, it could cause the inclusion of a reachable state. As described so far, IC3 would never attempt directly to block s since it only generalizes from backward reachable states. Yet blocking s, rather than joining with it, could enable finding an inductive subclause, thereby helping the generalization procedure produce stronger clauses and potentially shortening the way to a proof.

Listing 2. Proposed generalization procedure.

```
void MIC(q: cube ref, i: level):                          1
  MIC(q, i, 1)                                            2
                                                          3
void MIC(q: cube ref, i: level, d: recDepth):             4
  foreach literal l in q:                                 5
    q̂ := q \ l                                            6
    if ctgDown(q̂, i, d):                                  7
      q := q̂                                              8
                                                          9
bool ctgDown(q: cube ref, i: level,                       10
                          d: recDepth):                   11
  ctgs := 0                                               12
  while true:                                             13
    if I ⇏ ¬q:                                            14
      return false                                        15
    if F_i ∧ ¬q ∧ T ⇒ ¬q′:                                 16
      return true                                         17
    with (F_i ∧ ¬q)−state s:                              18
      if d > maxDepth:                                    19
        return false                                      20
      if ctgs < maxCTGs and i > 0 and                     21
            (I ⇒ ¬s) and (F_{i−1} ∧ ¬s ∧ T ⇒ ¬s′):         22
        ctgs := ctgs + 1                                  23
        for j := i to k do:                               24
          if F_j ∧ ¬s ∧ T ⇏ ¬s′:                           25
            break                                         26
        MIC(s, j − 1, d + 1)                              27
        clauses(F_j) := clauses(F_j) ∪ ¬s                 28
      else:                                               29
        ctgs := 0                                         30
        q := q ⊔ s                                        31
```

A generalization procedure that addresses CTGs is presented in Listing 2. Similarly to down, ctgDown first checks whether $\neg q$ is inductive (lines 14–17). However, if it is not inductive, it does not immediately join q with the discovered predecessor s. Rather, it attempts to block s at level i by proving it inductive relative to F_{i-1} (line 22). If this attempt succeeds, it tries to block it at higher levels (lines 24–26). It then strengthens the clause at the highest level relative to which it was found to be inductive by applying MIC (line 27). (Again, notice that cubes are passed by reference, so that when MIC returns, the cube s may be significantly expanded.) Having addressed one cause for the non-inductiveness of $\neg q$, ctgDown returns its attention to q.

To maintain its focus on the main goal of strengthening $\neg q$, ctgDown considers at most maxCTGs CTGs between joins (line 21). If the limit is exceeded or a CTG is not found to be inductive, the CTG is joined with q (line 31). New states brought in as a result of the join present an opportunity to explore behaviors farther from the error, so ctgDown resets the number of allowable CTGs to maxCTGs (line 30).

Since ctgDown calls MIC, the version of MIC associated with ctgDown monitors the recursion depth through its d parameter. The recursion depth is initialized by the wrapper function

to 1 (line 2) and updated by the call to MIC from ctgDown (line 27). At a recursion depth of 1, ctgDown examines CTGs that are encountered during generalization of a CTI-induced clause. For larger depths, an encountered CTG is one that interferes with the generalization of a CTG-induced clause. A limit, maxDepth, limits the effort spent on addressing CTGs of CTG-induced clauses. When this limit is exceeded (line 19), CTGs are not examined, and joins are disabled; instead, ctgDown fails immediately if $\neg q$ is not inductive (line 20).

B. Discussion

The recycled limit on handling CTGs results in an interesting pattern of state exploration. IC3 itself explores, through its priority queue, the state space in an explicit manner backward from the error. Let s be such a state: s can reach the error in a relatively few number of steps. If IC3 is forced to consider a predecessor of s, then it is known that the predecessor, too, can reach the error. In contrast, when MIC is applied to s, the first step is to drop a literal, enlarging the represented state set. In ctgDown, up to maxCTGs times, predecessors of the enlarged cube are then considered as CTGs. They are likely to be backward reachable; they are also likely to be about as close to an error as s is[1].

Eventually maxCTGs is exhausted, forcing a join. Predecessors to the enlarged cube are then considered as CTGs. These predecessors are less likely to be backward reachable but more likely to be "farther" from an error than s. Deep backward reachable states may be particularly valuable. This cycle can continue for several iterations, each iteration exploring states that are increasingly far from the error but at the cost of being increasingly likely not to be able to reach the error. Further iterations of dropping literals by MIC add layers of likelihoods of depth and backward reachability to the state exploration.

Another behavior worth noting is that ctgDown can fail more softly than down. When down fails, the only gained information is that the dropped literal is actually required. In contrast, ctgDown may successfully handle some CTGs on the way to failing to prove the inductiveness of the given cube. These CTG-derived lemmas could well prove useful in addressing the overall model checking problem.

In early attempts at considering CTGs, a scheme that delayed the handling of CTGs as much as possible was investigated. Rather than prioritizing the direct handling of CTGs over joining with them, it aggressively joined and only handled CTGs upon failure. If $\neg q_j$ failed initiation, the last-encountered CTG s was addressed directly. Successful elimination of s would enable the reconsideration of q_{j-1}; failure would cause the CTG leading from q_{j-2} to q_{j-1} to be addressed instead, and so on. This version was inferior to ctgDown, possibly because too much effort was put into addressing states that were either not actually backward reachable or too removed from the original CTI to be relevant to the generalization effort. ctgDown explores CTGs in an outwardly expanding set from the error; the unsuccessful variant explored CTGs in an inwardly contracting set.

While these explanations assume characteristics of a state space that need not hold for a given model, they offer an intuitive motivation for using ctgDown instead of down: with some trade-offs, it jumps to deep states, complementing IC3's conservative top-level behavior. Section V compares, empirically, the behavior of IC3 with down versus with ctgDown.

IV. Results

In this section IC3 with ctgDown is compared empirically to existing standard implementations of IC3. The new procedure was implemented within the IC3 engines in IImc v1.3 (upcoming release) [3] and in ABC `vbb0deac` (Apr 3, 2013) [4]. The implementations of ctgDown differ from the pseudocode of Listing 2 in the following respects:

- In the IImc implementation, the consecution call in line 26 was implemented as a call to down. This change enables blocking a CTG at a (higher) level at which its negation is not inductive but contains an inductive subclause. The experiments are inconclusive with regards to which version is better.
- In the ABC implementation, the CTG cube is expanded through ternary simulation before it is checked for inductiveness (line 18) [8].

ABC's standard implementation of IC3 does not employ down in its generalization procedure; in particular, it never joins. However, the implementation of ctgDown includes joining. Experiments (whose details are not reported here) show that a variant of ctgDown in which joining was disabled is inferior to full ctgDown.

Hence, experiments with IImc compare the effects of ctgDown against down, while experiments with ABC compare the effects of ctgDown against ABC's generalization procedure.

The following parameter values were used in the experiments for both implementations of ctgDown: $\mathsf{maxDepth} = 1$ and $\mathsf{maxCTGs} = 3$[2].

The benchmark suite was gathered from the HWMCC 2010–2012 benchmarks—with one exception. Backward-encoded BEEM models (distinguished by the names `beem*ibj`) were replaced with their corresponding "functional" versions, also available from [5]. The backward encoding of these models involves two features[3]:

1) Serial exists-step transition relation [9]: this feature adds "shortcut" transitions to the state graph.
2) Reverse relational encoding: the transition relation is inverted, and the initial states are swapped with the bad states. The latch updates are directly from primary inputs, and a `valid` bit is added to track whether a state is backward reachable in the original design.

[1] While there are models for which this assumption is invalid, the fraction of state bits of a large digital system that changes at each clock cycle is often less than one tenth. This fraction supports the view that similarity between states decreases with their distance in the state graph.

[2] Generally, small values for maxCTGs (2–5) gave the best performance. For higher values, IC3 tended to derive too many clauses.

[3] See http://fmv.jku.at/aiger/README.beemaigs for details.

IImc has a "reverse" option to invert the transition relation and exchange the initial and error states. With this option, IImc typically works better on reverse-encoded models and worse on forward-encoded ones. A possible explanation is that a clause is a natural logical means of describing a design intention; moreover, conjunctions of clauses capture local arguments. In contrast, disjunctions of cubes—which is what IC3 produces from the forward perspective on reverse-encoded designs—do not. With both the functional and the backward encodings of these models available, one would never choose to use the backward encoding with IC3. Conclusions drawn from data based on such benchmarks are misguided.

As a preprocessing step, IImc's `sr` simplification tactic was applied to each benchmark. Then, IImc and ABC with and without the new generalization procedure were run on the simplified benchmarks only invoking their IC3 engines. No other features of IImc or ABC—e.g., multi-threading, other proof engines, or more powerful simplification techniques—were used. Each benchmark was run for up to 900 seconds. To account for variability, each benchmark was run five times with different random seeds. The experiments were performed on two identical machines with four 2.80 GHz Intel cores and 9 GBs of memory. The full results can be found at http://vlsi.colorado.edu/fmcad13.

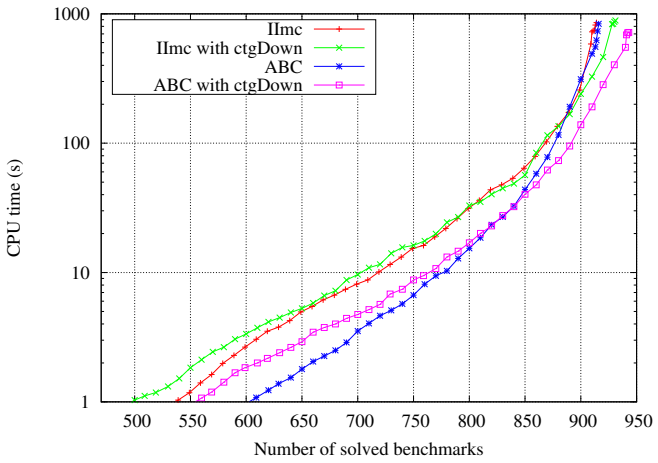


Fig. 2. Cactus plot comparing the performance of IImc and ABC with and without ctgDown.

A comparison between the performance of IC3 with and without ctgDown is presented in Figures 2–4. Figure 2 shows cactus plots for IImc and ABC and Figures 3 and 4 show scatter plots. All the plots use the results of the median runs.

For the easier models, the use of ctgDown does not typically reduce CPU time (Figures 3 and 4). An exception is the effect on the run times of failing properties with IImc.

Detailed results by benchmark family are presented in Table I. Benchmark families with at least 60 benchmarks are listed separately. The remaining benchmarks are included in the "other" category. The "Solved" columns show the minimum, median, and maximum number of solved instances over the five runs. The "Time" columns reports the median CPU time in

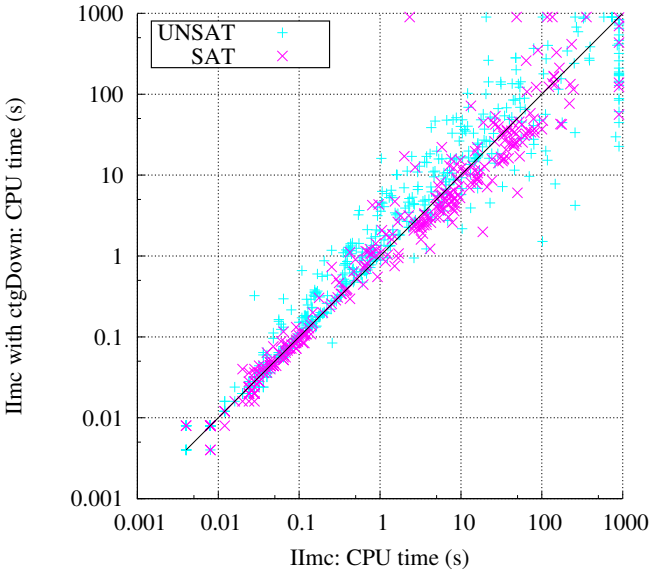


Fig. 3. IImc scatter plot

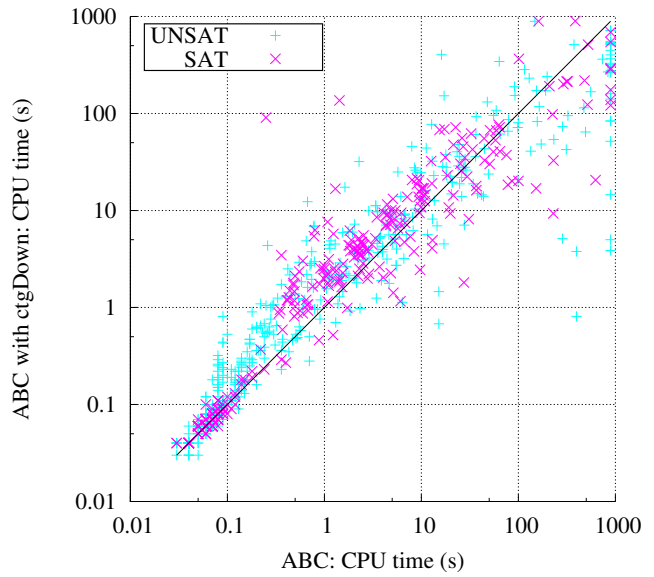


Fig. 4. ABC scatter plot

seconds. Overall, IImc and ABC with ctgDown solve 17 and 27 more instances, respectively, than their standard counterparts[4]. The same trend was observed when the timeout was increased to one hour: IImc and ABC solved 17 and 24 more instances respectively.

V. Analysis of IC3's Behavior

An observed weakness in IC3 with down is that on some models, it handles long chains of states explicitly rather than symbolically. ctgDown is intended to address this weakness

[4]Since the median is used, the sum of the gains for the individual families is not necessarily equal to the overall gain.

TABLE I
DETAILED RESULTS BY BENCHMARK FAMILY.

		IImc					ABC				
		Standard		With ctgDown			Standard		With ctgDown		
Family	Size	Solved	Time (s)	Solved	Gain	Time (s)	Solved	Time (s)	Solved	Gain	Time (s)
139	99	98/99/99	2524	99/99/99	0	1230	99/99/99	701	99/99/99	0	754
6s	120	18/19/22	93466	19/21/22	2	94211	19/23/24	88401	27/30/31	7	82941
beem	86	46/48/49	38149	47/50/51	2	39594	50/51/53	34098	54/56/57	5	31191
bob	149	121/122/125	25804	120/120/122	*(2)*	28679	122/123/124	24292	122/124/127	1	24083
intel	60	22/23/23	35004	29/30/31	7	31153	23/23/23	35665	25/26/27	3	34249
pdt	350	330/331/332	19291	336/336/337	5	15469	327/329/329	22162	333/333/333	4	18120
other	280	270/271/272	11947	272/274/275	3	11463	269/270/271	12591	272/274/274	4	10359
Total	1144	910/913/917	226790	924/930/932	**17**	222460	914/916/919	218906	936/943/944	**27**	201417

by accelerating IC3's exploration of deep backward reachable states while still maintaining its characteristic focus on the property. It attempts to achieve this objective by considering CTGs. As discussed, CTGs interfere with generalizing from CTIs and so are worthwhile candidates for blocking with generalization—although they need not be backward reachable. This section presents an analysis that, through measuring several metrics, suggests that ctgDown achieves its intended behavior. It highlights differences in the behavior of IC3 with the standard (down) and improved (ctgDown) generalization procedures. The data in this section were collected from IImc's IC3 runs. Data collected from ABC's runs also support the observations made.

Data points for scatter plots in this section are divided into two categories: those for which IC3 performs better with ctgDown, marked by a $\times$ in the plots, and those for which IC3 performs better with down, marked by a $+$.

The first experiment compares the average distances of CTGs and CTIs from an error. To measure the depths of CTGs, exact BDD-based backward reachability is performed; the resulting "onion rings" can be used to compute the depth of a given state. Of the CTGs handled in these experiments, 42% were backward reachable. For the depths of the CTIs, the length of the chain through which a CTI was found provides an upper bound on its actual backward depth. Figure 5 shows a plot for the average CTI depth against the average CTG depth for the 294 benchmarks for which the preliminary BDD-analysis managed to complete within 12 hours. The plot confirms that CTGs are typically deeper than CTIs—sometimes by several orders of magnitude. The plot also indicates that ctgDown helps in the cases where IC3 is forced to explore deep CTIs.

Next, several metrics of IC3 runs were analyzed to understand when the proposed generalization procedure helps or harms the performance of IC3. The metrics are the maximum length of traces from states in the priority queue to an error; the average size of derived clauses; the convergence level, i.e., the level at which a proof or a counterexample is found; and the average number of clauses derived per level.

Plots comparing IC3 with and without the proposed generalization procedure on the four metrics are shown in Figures 6a–6d. The same information is presented with box-and-whisker plots in Figure 6e with the ratio of each metric with ctgDown

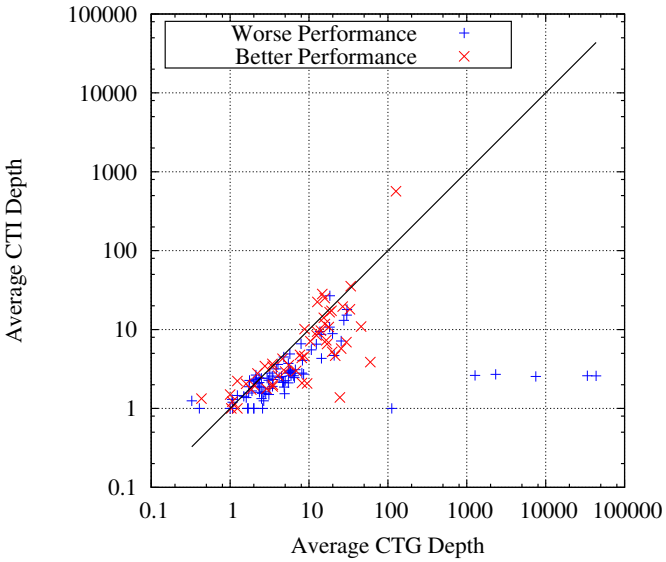


Fig. 5. A comparison between the depths of CTIs and CTGs.

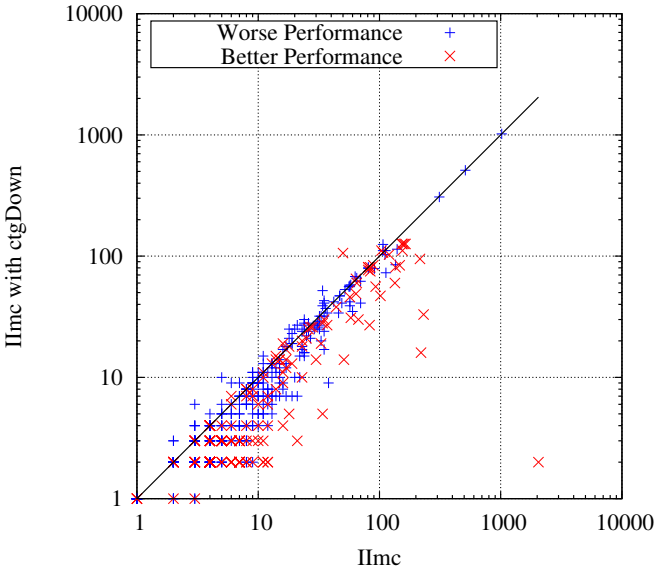
to without.

Figure 6a indicates a significant reduction in the depth of the explicit search performed by IC3 when ctgDown is used. Statistics indicate an average reduction of 22.3% in the depth of IC3's explicit search over all benchmarks. A higher reduction in the depth of the search often indicates better performance for IC3. This is confirmed by the non-overlapping notches in the box plot, which indicate a significant difference in the median depth ratios between cases with better and those with worse performance.

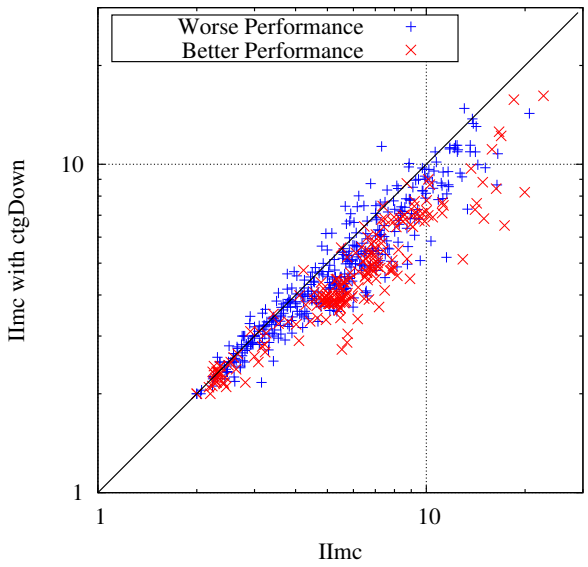
The point in the lower right corner of Figure 6a represents an extreme case in which IC3 with ctgDown proved the property with very little explicit backward search; with down, the depth of the priority queue went up to 2049.

Figure 6b points out ctgDown's ability to produce stronger CTI-induced clauses. Again, a stronger clause indicates improved performance. On average, ctgDown drops 14% more literals than down, which is statistically significant as indicated by the box plot.

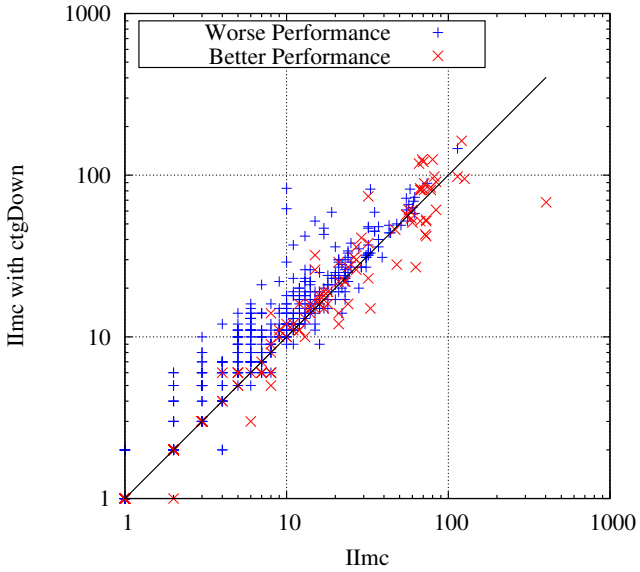
A characteristic of the new procedure is that it often increases the convergence level of IC3, as indicated in Figure 6c.



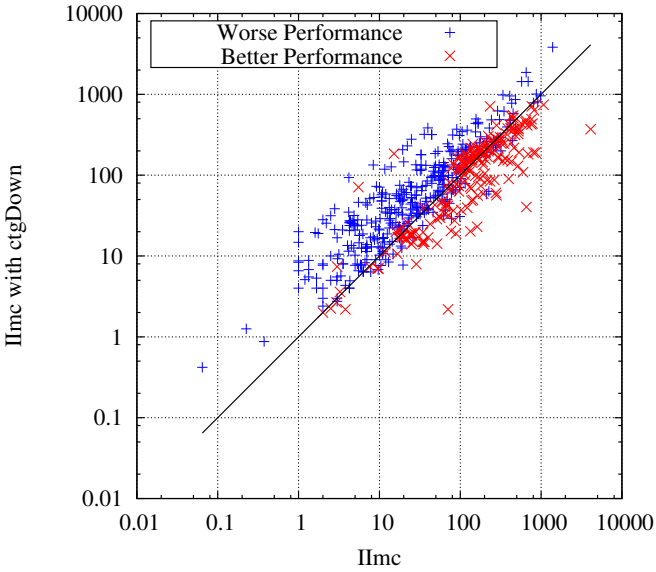
(a) Maximum depth of priority queue.



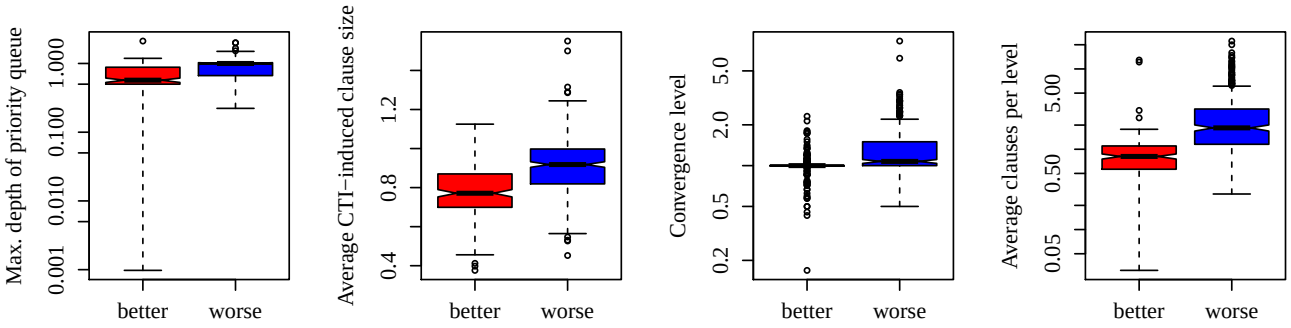
(b) Average CTI-induced clause size.



(c) Convergence level.



(d) Average clauses per level.



(e) Box plots for the ratios of the metrics shown in parts a–d.

Fig. 6. Analyzing the effects of ctgDown on the IImc runs.

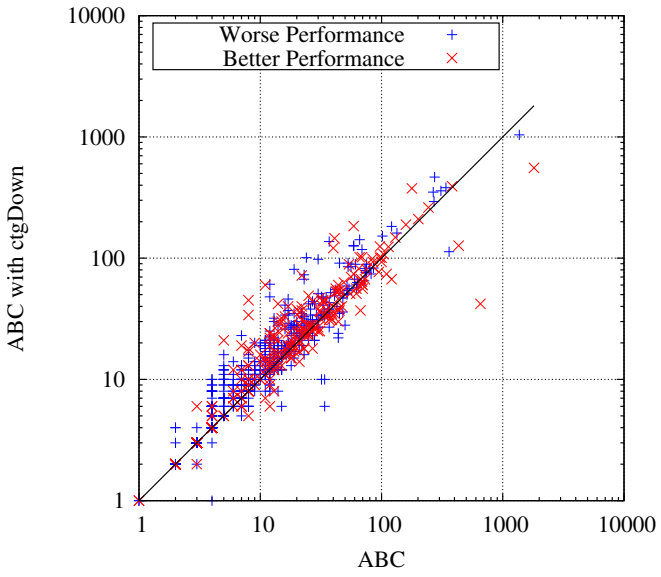


Fig. 7. Convergence level.

This potentially undesirable side effect is probably attributable to the aggressiveness of ctgDown in deriving clauses to block CTGs—which, again, need not actually be backward reachable. In contrast, the standard procedure only derives clauses in response to truly backward reachable states. A clause that blocks a forward reachable state is certainly not inductive and thus cannot appear in the final inductive strengthening. Such clauses can cause overstrengthening of the F_i's; then IC3 must propagate to higher levels in order to drop the clauses. Points to the far right in Figure 5 represent cases in which such behavior is exhibited. Although CTGs are much deeper than CTIs, the percentage of handled CTGs that are forward reachable is higher than average causing overstrengthening. Also, as Figure 6c shows, a higher convergence level is significantly correlated with worse performance. Similar observations hold for ABC with ctgDown as Figure 7 indicates. The box plot in Figure 6e shows that 75% of the runs in which ctgDown was beneficial did not increase the convergence level. In contrast, for 75% of the runs that did not benefit from ctgDown, the convergence level was higher. On the other hand, statistics indicate that the increase in convergence level only occurs for passing properties; for 75% of the failing properties, the convergence level isn't affected.

Points on the y-axis in Figure 6c correspond to benchmarks for which IC3 with down converges at level 1 while IC3 with ctgDown converges at higher levels. A characteristic behavior of IC3 with down is that clauses generated at level 1 are globally inductive until IC3 is forced to step back to level 0. Subsequently, generated clauses have the support of clauses generated relative to F_0 and thus need not be globally inductive. Aggressive handling of CTGs interferes with this initial behavior. A variant implementation was tried in which CTG handling was disabled until IC3 was forced to step back to level 0. IC3 with this variant ctgDown then converged at level 1 on these benchmarks; however, the performance difference across the benchmark suite was insignificant.

Finally, Figure 6d and the corresponding box plot indicate a clear correlation between the performance difference and the average number of clauses derived per level. An excessive number of clauses derived to block CTGs is often accompanied by longer runtimes.

VI. Related Work

Several improvements orthogonal to the generalization method presented here have been described for IC3. Ternary simulation [8] and SAT-based [10] methods of enlarging CTI cubes significantly improve running time. A scheme for integrating lazy abstraction with IC3 has also been developed [11].

VII. Conclusion

This paper presents an improved generalization procedure for IC3. Generalization is a key operation that lifts IC3 from explicit to symbolic analysis. Addressing states that impede generalization allows IC3 to deal with deep counterexamples to induction with less effort. The proposed procedure has been shown to significantly improve the performance of two independent implementations of IC3. While ctgDown achieves the objective of decreasing the depth of the explicit search, the impact on convergence level is mixed. Ongoing investigations seek to explain the interplay between the strength of lemmas, the convergence level, and the overall performance of IC3.

Acknowledgment

This work utilized the Janus supercomputer, which is supported by the National Science Foundation (award number CNS-0821794) and the University of Colorado Boulder. The Janus supercomputer is a joint effort of the University of Colorado Boulder, the University of Colorado Denver and the National Center for Atmospheric Research. Janus is operated by the University of Colorado Boulder.

References

[1] A. R. Bradley, "k-step relative inductive generalization," CU Boulder, Tech. Rep., Mar. 2010, http://arxiv.org/abs/1003.3649.

[2] ——, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Austin, TX, Jan. 2011, pp. 70–87, lNCS 6538.

[3] "URL: http://iimc.colorado.edu."

[4] "URL: http://www.eecs.berkeley.edu/~alanmi/abc/."

[5] "Hardware model checking competition. http://fmv.jku.at/hwmcc."

[6] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[7] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, Austin, TX, Nov. 2007, pp. 173–180.

[8] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property-directed reachability," in *Formal Methods in Computer Aided Design (FMCAD'11)*, Austin, TX, Nov. 2011, pp. 125–134.

[9] J. Dubrovin, T. Junttila, and K. Heljanko, "Exploiting step semantics for efficient bounded model checking of asynchronous systems," *Science of Computer Programming*, vol. 77, no. 10-11, pp. 1095–1121, 2012.

[10] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Formal Methods in Computer Aided Design (FMCAD'11)*, Austin, TX, Nov. 2011, pp. 135–143.

[11] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and SAT-based reachability for hardware model checking," in *Formal Methods in Computer-Aided Design (FMCAD'12)*, Oct. 2012.

Parameter Synthesis with IC3

Alessandro Cimatti Alberto Griggio * Sergio Mover Stefano Tonetta
Fondazione Bruno Kessler, Trento, Italy
{cimatti,griggio,mover,tonettas}@fbk.eu

***Abstract*—Parametric systems arise in different application domains, such as software, cyber-physical systems or tasks scheduling. A key challenge is to estimate the values of parameters that guarantee the desired behaviours of the system.**

In this paper, we propose a novel approach based on an extension of the IC3 algorithm for infinite-state transition systems. The algorithm finds the feasible region of parameters by complement, incrementally finding and blocking sets of "bad" parameters which lead to system failures. If the algorithm terminates we obtain the precise region of feasible parameters of the system.

We describe an implementation for symbolic transition systems with linear constraints and perform an experimental evaluation on benchmarks taken from the domain of hybrid systems. The results demonstrate the potential of the approach.

I. Introduction

Parametric systems arise in many application domains from real-time systems to software to cyber-physical systems. In these applications, the system is often part of a larger environment, and the designer has to define the system relative to some unknown parameters of the environment. The design of a robust system requires the verification to not rely on concrete values for the parameters but to prove the correctness of the system for a certain region of values. The use of parameters is fundamental in the early phases of the development, giving the possibility to explore different design choices. In fact, a parametric system represents a set of (non-parametric) systems, one for each valuation of the parameters.

A key challenge for the design of parametric systems is the estimation of the parameter valuations that guarantee the correct behavior of the system. Manual estimation of these values is time consuming and does not find optimal solutions for specific design problems. Therefore, a fundamental problem is to automatically synthesize the maximal region of parameter valuations for which the system satisfies some properties.

In this paper, we focus on the verification of invariant properties and how to extend the SMT-based algorithms to solve the synthesis problem. The general approach works by complement, building the set of "bad" parameter valuations. It relies on the enumeration of counterexamples violating the properties, extracting from the counterexample a region of bad parameter valuations by quantification of the state variables.

The novel contribution of this paper is a new synthesis algorithm based on IC3, one of the major recent breakthroughs in SAT-based model checking, and lately extended to the SMT case. The key idea of the synthesis algorithm is to exploit the features of IC3. First, IC3 may find a set of counterexamples consisting of a sequence of set of states $s_o, \ldots, s_k$, where each state in s_i is guaranteed to reach some of the bad states in s_k in $k-i$ steps; this is exploited in the expensive quantification of the state-variables, that can be performed on shortest, and thus more amenable, counterexamples. Second, the internal structures of IC3 allows our extension to be integrated in a fully incremental fashion, never restarting the search from scractch to find a new counterexample.

Various approaches already solve the parameter synthesis problem for several kind of systems, like infinite-state transition systems [4], timed and hybrid automata [10], [12], [9], [7], [1], [2]. The advantages of the new algorithm are that it synthesizes an optimal region of parameters (unlike [9], [1]), it is incremental and applies quantifier elimination only to small formulas (unlike [9], [7]), and it avoids computing the whole set of the reachable states (unlike [10], [12]).

We implemented the algorithm for symbolic transition systems with linear constraints and performed an experimental evaluation on benchmarks on timed and hybrid systems. We compared the approach with similar SMT-based techniques and with techniques based on the computation of the reachable states. The results show the potential of the approach.

II. Background

A. Transition Systems

A *transition system* S is a tuple $S = \langle X, I, T \rangle$ where X is a set of (state) variables, $I(X)$ is a formula representing the initial states, and $T(X, X')$ is a formula representing the transitions. In this paper, we shall deal with *linear rational arithmetic* formulas, that is, Boolean combinations of propositional variables and linear inequalities over rational variables. A *state* of S is an assignment to the variables X. A *path* of S is a finite sequence $s_0, s_1, \ldots, s_k$ of states such that $s_0 \models I$ and for all i, $0 \leq i < k$, $s_i, s'_{i+1} \models T$. Given a formula $P(X)$, the *verification problem* denoted with $S \models P$ is the problem to check if for all paths $s_0, s_1, \ldots, s_k$ of S, for all i, $0 \leq i \leq k$, $s_i \models P$. The dual problem is the *reachability problem*, which is the problem to find a path $s_0, s_1, \ldots, s_k$ of S such that $s_k \models \neg P$. $P(X)$ represents the "good" states, while $\neg P$ represents the "bad" states.

B. Parameter Synthesis

In parametric systems, besides the standard constants, the formulas can include also *parameters*, which are rigid symbols

* Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

ISBN 978-0-9835678-3-7/13.

with "unknown" values. Let U be the set of parameters. A *parameter valuation* is as assignment to the parameters. Given a formula ϕ and a parameter valuation γ, we denote with $\gamma(\phi)$ the formula obtained from ϕ by replacing each parameter in U with the assignment given by γ.

A *parametric transition system* S is a tuple $S = \langle U, X, I, T\rangle$ where U is the set of parameters, X is the set of variables, $I(U, X)$ is the initial formula, and $T(U, X, X')$ is the transition formula. Each parameter valuation γ induces a transition system $S_\gamma = \langle X, \gamma(I), \gamma(T)\rangle$.

Given a parametric transition system $S = \langle U, X, I, T\rangle$ and a formula $P(U, X)$, we say that a parameter valuation γ is *feasible* iff $S_\gamma \models \gamma(P)$. The *parameter synthesis problem* is the problem of finding a set $\rho(U)$ of feasible parameter valuations (i.e., for every $\gamma \in \rho$, $S_\gamma \models \gamma(P)$). A set of feasible parameter valuations $\rho(U)$ is *optimal* if it contains all the feasible parameter valuations.

C. IC3 *with SMT*

IC3 [3] is an efficient algorithm for the verification of finite-state systems, with Boolean state variables and propositional logic formulas. IC3 was subsequently extended to the SMT case in [5], [11]. In the following, we present its main ideas, following the description of [5]. For brevity, we have to omit several important details, for which we refer to [3], [5], [11].

Let S and P be a transition system and a set of good states as in §II-A. The IC3 algorithm tries to prove that $S \models P$ by finding a formula $F(X)$ such that: (i) $I(X) \models F(X)$; (ii) $F(X) \wedge T(X, X') \models F(X')$; and (iii) $F(X) \models P(X)$.

In order to construct an inductive invariant F, IC3 maintains a sequence of formulas (called *trace*) $F_0(X), \ldots, F_k(X)$ such that: (i) $F_0 = I$; (ii) $F_i \models F_{i+1}$; (iii) $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$; (iv) for all $i < k$, $F_i \models P$.

The algorithm proceeds incrementally, by alternating two phases: a blocking phase, and a propagation phase. In the *blocking* phase, the trace is analyzed to prove that no intersection between F_k and $\neg P(X)$ is possible. If such intersection cannot be disproved on the current trace, the property is violated and a counterexample can be reconstructed. During the blocking phase, the trace is enriched with additional formulas, that can be seen as strengthening the approximation of the reachable state space. At the end of the blocking phase, if no violation is found, $F_k \models P$.

The *propagation* phase tries to extend the trace with a new formula F_{k+1}, moving forward the clauses from preceding F_i's. If, during this process, two consecutive elements of the trace (called *frames*) become identical (i.e. $F_i = F_{i+1}$), then a fixpoint is reached, and IC3 terminates with F_i being an inductive invariant proving the property.

In the *blocking* phase IC3 maintains a set of pairs (s, i), where s is a set of states that can lead to a bad state, and $i > 0$ is a position in the current trace. New formulas (in the form of clauses) to be added to the current trace are derived by (recursively) proving that a set s of a pair (s, i) is unreachable starting from the formula F_{i-1}. This is done by checking the satisfiability of the formula:

$$F_{i-1} \wedge \neg s \wedge T \wedge s'. \tag{1}$$

If (1) is unsatisfiable, and s does not intersect the initial states I of the system, then $\neg s$ is *inductive relative to* F_{i-1}, and IC3 strengthens F_i by adding $\neg s$ to it[1], thus *blocking* the bad state s at i. If, instead, (1) is satisfiable, then the overapproximation F_{i-1} is not strong enough to show that s is unreachable. In this case, let p be a subset of the states in $F_{i-1} \wedge \neg s$ such that all the states in p lead to a state in s' in one transition step. Then, IC3 continues by trying to show that p is not reachable in one step from F_{i-2} (that is, it tries to block the pair $(p, i-1)$). This procedure continues recursively, possibly generating other pairs to block at earlier points in the trace, until either IC3 generates a pair $(q, 0)$, meaning that the system does not satisfy the property, or the trace is eventually strengthened so that the original pair (s, i) can be blocked.

A key difference between the original Boolean IC3 and its SMT extensions in [5], [11] is in the way sets of states to be blocked or generalized are constructed. In the blocking phase, when trying to block a pair (s, i), if the formula (1) is satisfiable, then a new pair $(p, i-1)$ has to be generated such that p is a cube in the *preimage of* s *wrt.* T. In the propositional case, p can be obtained from the model μ of (1) generated by the SAT solver, by simply dropping the primed variables occurring in μ. This cannot be done in general in the first-order case, where the relationship between the current state variables X and their primed version X' is encoded in the theory atoms, which in general cannot be partitioned into a primed and an unprimed set. The solution proposed in [5] is to compute p by existentially quantifying (1) and then applying an *under-approximated* existential elimination algorithm for linear rational arithmetic formulas. Similarly, in [11] a theory-aware generalization algorithm for linear rational arithmetic (based on interpolation) was proposed, in order to strengthen $\neg s$ before adding it to F_i after having successfully blocked it.

III. Parameter Synthesis with IC3

A. Solving the synthesis problem with reachability

A naive approach to synthetize the set of parameters $\rho(U)$ is to incrementally find the complement set $\beta(U)$ (thus, $\rho = \neg\beta$) of unfeasible parameter valuations, rephrasing the problem as a reachability problem for a transition system S_ρ and iteratively removing the counterexamples to $S_\rho \models P$.

More specifically, given the parametric transition system $S = \langle U, X, I, T\rangle$, the algorithm keeps an over-approximation $\rho(U)$, initially true, of the safe region. The encoding of S is the transition system $S_\rho = \langle X \cup P, I_\rho, T_\rho\rangle$ where $T_\rho = T \wedge \bigwedge_{p \in U} p' = p$ forces parameters to not change their value in the evolution of the system and $I_\rho = I \wedge \rho$ restricts the parameter valuations to the over-approximation.

At every iteration, a new parameter valuation is removed from ρ. The algorithm terminates if it proves that $S_\rho \models P$, and ρ is the solution to the synthesis problem.

[1] $\neg s$ is actually *generalized* before being added to F_i. Although this is foundamental for the IC3 effectiveness, we do not discuss it for simplicity.

This simple approach does not work in the context of infinite-state transition systems, where the possible number of counterexamples and the values of the parameters are infinite. For this reason, we need an algorithm that removes a set of parameters, instead of a single point.

B. Description of the synthesis algorithm with IC3

We embed a reasoning similar to the naive algorithm in IC3, exploiting its generalization of counterexamples and incrementality. The generalization avoids to explicitly enumerate the counterexamples, while incrementality allows to reuse all the clauses learned by IC3 across different safety checks.

Therefore, IC3 is used to prove that $S_\rho \models P$. If it is successful (recall that in the SMT extension, the problem is undecidable), we can conclude that ρ is a set of feasible parameters and, in particular, is optimal. Instead, if there exists a set of parameters such that $S \not\models P$, IC3 might find a counterexample to P. The counterexample is found in the blocking phase as a sequence $\pi := (s_0, 0), \ldots, (s_n, n)$, where $s_0 \models I'$, $s_n \models \neg P$ and for $0 < i < n-1$, $s_i \wedge T' \models s_{i+1}$. Possibly, π does not represent a single path of the system that reaches a violation, but a set of paths that reach $\neg P$. This is an intrisic feature of IC3, which generalizes the counterexamples to induction found in the blocking phase, trying to block set of states rater than a single state[2]. The state s_o represents a set of states that will eventually reach $\neg P$. Thus, we compute from s_0 a set of bad parameters $\beta_{s_o}(U)$ that will eventually reach s_n: $\beta_{s_o}(U) := \exists X.s_o(U, X)$. We rely on a quantifier elimination procedure to get a quantifier-free formula for β_{s_o}.

The algorithm refines its conjecture about the unfeasible parameters of the system. Let $\beta' := \beta \vee \beta_{s_o}$ and $\rho' := \rho \wedge \neg\beta_{s_o}$ be the new approximations of unfeasible and feasible regions of parameters. We have to prove that $S_{\rho'} \models P$. We perform the verification incrementally, reusing all the frames of IC3. Since $\rho' := \rho \wedge \neg\beta_{s_o}$, we have that $S_{\rho'} = \langle X \cup P, I_\rho \wedge \neg\beta_{s_o}, T_\rho \rangle$.[3] Thus, we incrementally encode $S_{\rho'}$ strengthening the initial condition and the transition relation used in the algorithm, and also strengthening the first frame kept by the IC3 algorithm (i.e. $F_0 := F_0 \wedge \neg(\beta_{s_o})$). The strengthening of F_0 removes the state s_o from I (possibly blocking also other bad states).

Since S_ρ is an overapproximation of $S_{\rho'}$, the invariant mantained by IC3 ($F_0 = I$, $F_i \models F_{i+1}$, $F_i \models P$, $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$) holds for the problem $S_{\rho'} \models P$.

From this point, we rely on the usual behaviour of IC3, which tries to block $(s_1, 1)$ with the strengthened frame F_0. The algorithm terminates if either P is proved or the F_0 becomes unsatisfiable, showing that ρ is empty.

Theorem 1: Given a parametric transition system $S = \langle U, X, I, T \rangle$ and a formula $P(X)$, $\rho(U) :=$ ParamIC3(U, I, T, P) is the *optimal set* of feasible parameter valuations.

[2]We follow the IC3 formulation of [8], which shows that IC3 can find a set of counterexamples, improving its performance. Moreover, in the SMT-based IC3 [5] the approximate pre-image computes a set of states.

[3]We add $\neg\beta_{s_o}$ also to T_ρ, since it is an inductive invariant of $S_{\rho'}$.

For lack of space, the proof is available in a technical report (http://es.fbk.eu/people/mover/fmcad13_ext.pdf).

C. Optimizations

We presented a version of the algorithm which computes a region of bad states β_{s_o} only from the initial states of $\pi := (s_0, 0), \ldots, (s_n, n)$. However, this is only one of the possible choices, since more general regions of bad parameters can be found considering each s_i in π. In fact, β_{s_o} is one of the extreme cases, while the other one is $\beta_n(U) := \exists X.(BMC_n)$, which encodes the set of all the parameters that may reach $\neg P$ in n steps, where BMC_n denotes $I^0 \wedge \bigwedge_{i=0}^{n-1} T^i \wedge \neg P^n$. However, the cost of eliminating the quantifiers grows as well, and it might in fact become impractical. In principle, one may consider the intermediate cases β_{s_i} (that is, the reachability of one of the intermediate states s_i in π) to trade the generality of the result with the cost of the quantifier elimination. Furthermore, we notice that for soundness we do not need the precise set β_{s_i}, but we can consider its under-approximations, since this still guarantees to remove only bad parameters valuations. As an advantage, in this case the quantifier elimination problems are easier to solve and are more general than β_{s_o}. In practice, we use an heuristic, which we describe in the next Section, that combines the precise and the under-approximated approach, enabling us to find a trade-off between generality and the cost of quantifier elimination.

IV. Experiments

We have implemented the algorithm described in the previous section on top of the fully symbolic SMT-based IC3 of [5]. The tool uses MathSAT [6] as backend SMT engine, and works on transition systems with linear arithmetic constraints.

Evaluation. Our evaluation consists of three parts. In the first, we compare our implementation (called ParamIC3 in what follows) with the approach described in [7], in order to evaluate the viability of our technique when compared to other SMT-based solutions. For this, we have implemented the algorithm described in [7] using our "regular" SMT-based IC3 implementation as the backend engine for reachability checking (called Iterative-Block-Path(IC3) in what follows). We remark that the tool of [7] was based only on Bounded Model Checking (BMC), and exploited domain-specific information for computing the maximum needed bound, which is not available in our more general context.

In the second part, we evaluate the effectiveness of the optimizations described in the previous section, by comparing the default heuristic used by ParamIC3, using both the full counterexample path π and its initial state $(s_0, 0)$ for blocking bad regions of parameters, with the basic strategy using only $(s_0, 0)$ (called ParamIC3-basic in the following). In particular, the default heuristic used by ParamIC3 works as follows. At the beginning, only initial states $(s_0, 0)$ of counterexample paths are used to block bad regions of parameters. If the algorithm starts enumerating too many bad regions, it starts exploiting also full paths π, by computing the bad region $\beta_k^\pi(U) = \exists X.BMC_k^\pi$, where k is the length of π, and BMC_k^π

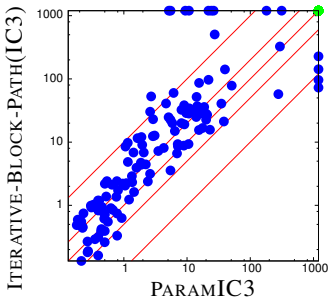


Fig. 1. Run time comparison (sec.) between PARAMIC3 and ITERATIVE-BLOCK-PATH(IC3).

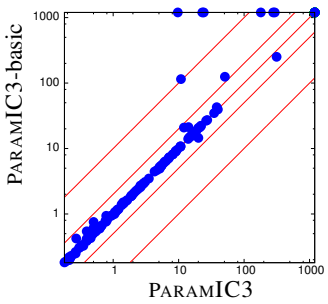


Fig. 2. Run time comparison (sec.) between PARAMIC3 and PARAMIC3-basic.

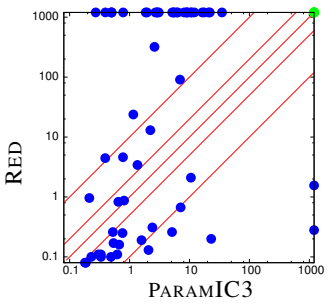


Fig. 3. Run time comparison (sec.) between PARAMIC3 and RED.

is the formula encoding all the counterexample traces of length k where the values for the Boolean variables are the same as in π, similarly to what is done in [7]. The computation of β_k^π is aborted if it becomes too expensive[4] , in order to control the tradeoff between the quality of the obtained bad region and the cost of performing quantifier elimination.

Finally, in the third part of our evaluation, we compare PARAMIC3 against RED [12], a state-of-the-art tool for parameter synthesis for linear-hybrid automata.

Benchmarks. We have selected benchmarks used in previous work on parameter synthesis for hybrid systems. Most of them come from the suite of RED. We have a total of 92 instances from 13 different families. All the instances, the scripts and the tools used for reproducing our experiments are available at http://es.fbk.eu/people/mover/fmcad13.tar.gz. For the first two parts of our evaluation, we have experimented with two different ways of encoding linear hybrid automata into symbolic transition systems, resulting in a set of 192 instances. For the comparison with RED, we picked the encoding giving the best overall performance for PARAMIC3.

Results. We have run our experiments on a cluster of Linux machines with a 2.27GHz Xeon CPU, using a timeout of 600 seconds and a memory limit of 3Gb for each instance. Figures 1–3 show the scatter plots that compare the total run time (in seconds) of the different techniques. From the plots, we can make the following observations. (i) Our new algorithm is clearly superior to the technique of [7], both in number of completed instances and in execution time. Overall, PARAMIC3 successfully solves 5 more instances than ITERATIVE-BLOCK-PATH(IC3), and it is almost always faster. We remark that both algorithms use the same implementation of IC3 as backend, run with the same options. (ii) Our heuristic for using full counterexample paths π for blocking bad regions of parameters pays off for harder problems. With it, PARAMIC3 solves 6 more instances which were previously out of reach, without any overhead for the other instances. (iii) The comparison with RED shows that our technique is very promising. Although there is no clear winner, there are more instances for which PARAMIC3 outperforms RED than the converse. In general, the two tools seem to be somewhat complementary. We remark that RED is specialized for timed and linear-hybrid automata and that most of the benchmarks we used come from its suite, whereas PARAMIC3 works for arbitrary transition systems and it is not tuned for linear hybrid systems in any way.

[4]We currently use a cutoff value on the number of elementary operations in the quantifier elimination module of MATHSAT for this.

V. CONCLUSIONS AND FUTURE WORK

We proposed a new algorithm based on IC3 for synthesizing an optimal region of parameter valuations guaranteeing the satisfaction of an invariant property. The algorithm exploits the features of IC3 to incrementally remove sets of bad parameter valuations and to reduce the cost of expensive quantifier elimination operations by performing them on small formulas. Our experimental results show that the new synthesis algorithm performs better than similar SMT-based techniques and is complementary to other techniques based on the computation of the reachable states. In the future, we plan to improve the algorithm by better exploiting the structure of the problem, to evaluate it in other domains such as software, and to apply it in the context of modular component-based verification.

REFERENCES

[1] É. André and U. Kühne. Parametric analysis of hybrid systems using HyMITATOR. In *iFM*, 2012.
[2] G. Behrmann, K. Guldstrand Larsen, Jacob Illum Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In *FORMATS*, 2005.
[3] A. Bradley. Sat-based model checking without unrolling. In *VMCAI*, 2011.
[4] R. Bruttomesso, A. Carioni, S. Ghilardi, and S. Ranise. Automated analysis of parametric timing-based mutual exclusion algorithms. In *NFM*, 2012.
[5] A. Cimatti and A. Griggio. Software Model Checking via IC3. In *CAV*, 2012.
[6] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, 2013.
[7] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *RTSS*, 2008.
[8] N. Eén, A. Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134, 2011.
[9] G. Frehse, S. Jha, and B. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC*, 2008.
[10] T. Henzinger and P. Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems*, 1994.
[11] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
[12] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with bdd-like data-structures. *IEEE Trans. Software Eng.*, 31(1), 2005.

Generalized Counterexamples to Liveness Properties

Gadi Aleksandrowicz, Jason Baumgartner, Alexander Ivrii, Ziv Nevo IBM Corporation

***Abstract*—We consider *generalized counterexamples* in the context of liveness property checking. A generalized counterexample comprises only a subset of values necessary to establish the existence of a concrete counterexample. While useful in various ways even for safety properties, the length of a generalized *liveness* counterexample may be exponentially shorter than that of a concrete counterexample, entailing significant potential algorithmic benefits.**

One application of this concept extends the k-LIVENESS proof technique of [1] to enable failure detection. The resulting algorithm is simple, and poses negligible overhead to k-LIVENESS in practice. We additionally propose dedicated algorithms to search for generalized liveness counterexamples, and to manipulate generalized counterexamples to and from concrete ones. Experiments confirm the capability of these techniques to detect failures more efficiently than existing techniques for various benchmarks.

I. Introduction

It is well-known that counterexamples are often redundant, containing many values that are irrelevant to the failure exhibited therein. The process of eliminating unnecessary values from a trace is referred to as *generalization*, and has numerous benefits. For example, the elimination of irrelevant values facilitates manual and automated debugging [2], and improves the effectiveness of counterexample-guided abstraction refinement [3].

This paper focuses upon counterexamples to liveness properties. For finite systems, such counterexamples may efficiently be represented as lasso-shaped traces consisting of a prefix and a loop suffix exhibiting a state repetition which can be infinitely unrolled. The length of a lasso is the sum of the prefix and suffix lengths. A unique benefit of generalizing a liveness counterexample is that it may shorten the lasso length – possibly exponentially so – if the set of state variables comprising a state repetition is reduced during generalization.

Example 1: Let q, x, y be Boolean signals whose initial and next-state behaviors are determined as follows: $q_0 = 1$, $x_0 = 0$, $y_0 = 0$, $q' = (q \wedge x) \vee (\neg q \wedge y)$, $x' = q \wedge y$, $y' = \neg x$. Consider liveness property FGq, specifying that on every trace q must eventually become true forever. A counterexample would illustrate $q = 0$ at least once in its loop suffix, for example $(q, x, y) = (1, 0, 0) \rightarrow (0, 0, 1) \rightarrow (1, 0, 1) \rightarrow (0, 1, 1) \rightarrow (1, 0, 0)$ of length 4. Note that $(q = 1) \wedge (x = 0) \Rightarrow (q' = 0) \wedge (y' = 1)$ and $(q = 0) \wedge (y = 1) \Rightarrow (q' = 1) \wedge (x' = 0)$. This illustrates a generalized counterexample: $(1, 0, \cdot) \rightarrow (0, \cdot, 1) \rightarrow (1, 0, \cdot)$ of length 2.

Example 2: We may modify q' from Example 1 to $q' = (q \wedge x \wedge (cnt = 0)) \vee (\neg q \wedge y)$, where cnt is an n-bit cyclic counter. Now the minimal concrete counterexample has length 2^n, while the generalized counterexample from Example 1 is still valid.

Example 3: One may argue that cnt is sequentially unobservable in Example 2 because $q \wedge x \equiv 0$, hence a transformation-based approach may enable the detection of an adequate short counterexample [4]. We may modify this example to $x' = (q \vee i) \wedge y$, where i is a nondeterministic input. Now cnt becomes observable, precluding a direct application of transformation-based methods. However, the generalized counterexample is still valid since both transitions $(1, 0, \cdot) \rightarrow (0, \cdot, 1)$ and $(0, \cdot, 1) \rightarrow (1, 0, \cdot)$ can be achieved *for some value of the inputs*, here $i = 0$ for transition $(0, \cdot, 1) \rightarrow (1, 0, \cdot)$.

We show that, surprisingly, the traces produced by the underlying safety model checker of k-LIVENESS [1] are often sufficient to witness a counterexample. Furthermore, in many cases the traces which do not exhibit a counterexample may be manipulated using our techniques to yield valid counterexamples.

II. Preliminaries

We represent a finite state system S as a tuple $\langle i, x, I(x), T(i, x, x') \rangle$, which consists of primary inputs i, state variables x, predicate $I(x)$ defining the initial states, and predicate $T(i, x, x')$ defining the transition relation. Next-state variables are denoted as x'. We assume that T 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 propagation – i.e., using Boolean or three-valued simulation.

State variables and their negations are called *literals*, and disjunctions (conjunctions) of literals are called *clauses* (*cubes*). A *state* is a Boolean assignment to all of x. A *generalized state* is an assignment to a subset of x, representing a set of states. We denote concrete states by s and generalized states by t throughout the paper.

ISBN 978-0-9835678-3-7/13.

Definition 1: Given two generalized states t_0 and t_1, we say that t_0 is a *predecessor* of t_1 if for every concrete state $s_0 \in t_0$ there exists a concrete state $s_1 \in t_1$ and an input i_0 such that $(i_0, s_0, s_1') \models T$.

Definition 2: We say that t_0 is a *concrete* predecessor of t_1 if t_0 is a concrete state.

Note that the definition of a predecessor is not symmetric, and it does not require that every state in t_1 is reachable from a state in t_0. For practical purposes we need more restricted notions of a predecessor.

Definition 3: Given two generalized states t_0 and t_1 and input i_0, we say that t_0 is an *implying* predecessor of t_1 with respect to i_0 if $t_0 \wedge i_0 \wedge T \wedge \neg t_1'$ is unsatisfiable.

Definition 4: We say that t_0 is a *propagating* predecessor of t_1 with respect to i_0 if t_0 and i_0 imply t_1' by propagation.

From these definitions, each concrete predecessor is also propagating with respect to some input, and each propagating predecessor with respect to i_0 is also implying with respect to i_0. We omit explicit input references when they are clear from context.

Example 4: Consider $x' = x \wedge (y \oplus i)$, where x and y are state variables and i is an input. Then $x = 1 \wedge y = 1$ is an implying predecessor of $x = 1$ with respect to $i = 0$. Further, $x = 1$ is predecessor of $x = 1$ but cannot be an implying predecessor since there is no value of i which works for all y.

Definition 5: A *concrete trace* is a sequence of concrete states $\langle s_0, \ldots, s_n \rangle$ such that $s_0 \models I$, and for each $0 \leq k < n$, s_k is a concrete predecessor of s_{k+1}.

Definition 6: A *generalized trace* is a sequence of generalized states $\langle t_0, \ldots, t_n \rangle$ such that t_0 contains an initial state, and for each $0 \leq k < n$, t_k is a predecessor of t_{k+1}.

We say that *concretizing* a state t is the process of adding literals to t, and *generalizing* t is the process of removing literals from t.

A. Generalized Counterexamples to Liveness

In the spirit of [1] we consider liveness properties given in the form FGq. More general liveness properties (and fairness constraints) may be reduced to this form using additional logic. Furthermore, since the validity of $FG(Xq)$ is equivalent to the validity of FGq, we can assume that q itself is a state variable.

Definition 7: A *concrete counterexample* to FGq is a concrete trace $\langle s_0, \ldots, s_n \rangle$ and an index m with $0 \leq m < n$, such that **(1)** $s_m = s_n$, and **(2)** $\exists k \in [m..n]$ with $s_k \implies \neg q$.

Thus $s_0, \ldots, s_{m-1}$ corresponds to the lasso prefix, and $s_m, \ldots, s_n$ corresponds to the loop suffix, with cycle s_k exhibiting $\neg q$. Note that $s_m = s_n$ implies that s_n is a concrete predecessor of s_{m+1}, hence the loop can be infinitely unrolled.

Definition 8: A *generalized counterexample* to FGq is a generalized trace $\langle t_0, \ldots, t_n \rangle$ and an index m with $0 \leq m < n$, such that **(1)** $t_m \implies t_n$, and **(2)** $\exists k \in [m..n]$ with $t_k \implies \neg q$.

Note that we do not require that $t_m = t_n$, but rather that t_n is more concrete than t_m.

Examples 1-3 illustrate that the length of a generalized counterexample to FGq may be exponentially shorter (with respect to netlist size) than that of a concrete counterexample. Theorem 1 will demonstrate that the former implies the existence of the latter. Because a generalized counterexample may be exponentially shorter than a concrete one, in cases it may be easier to detect a generalized counterexample, which motivates the algorithms in Sections IV and V.

In practice, a generalized counterexample may actually be more informative and easier to debug since it more clearly illustrates the "essential" reason for the failure. Similarly, it is often undesirable in practice that a liveness counterexample on a reduced netlist (after cone-of-influence, redundancy removal, ...) be extended to a possibly exponentially-longer unreduced trace merely to ensure a state repetition over irrelevant logic.

III. Trace Manipulation Algorithms

A. Trace Concretization

Given a generalized trace, we may fully or partially concretize it using Algorithm 1. $ConcretizeInitial(t_0)$ returns a concretization of t_0 which still contains a state in I, which may be computed with a satisfiability query. $ConcretizeForward(\tilde{t}_k, t_{k+1})$ returns a concretization of t_{k+1} with $\tilde{t}_k$ as its predecessor. If $\tilde{t}_k$ is a propagating predecessor of t_{k+1}, we can use three-valued simulation to implement $ConcretizeForward$, using an unknown X value for any state variable not in $\tilde{t}_k$ and assessing which state variables attain fixed values in t_{k+1}. Alternatively, we can use a satisfiability query: if $\tilde{t}_k$ is an implying predecessor of t_{k+1} with respect to some i_k, for each state variable x not in t_{k+1} we can consider the query $\tilde{t}_k \wedge i_k \wedge T \wedge x'$. If this query is unsatisfiable, $x = 0$ can be added to t_{k+1}. Similarly, if $\tilde{t}_k \wedge i_k \wedge T \wedge \neg x'$ is unsatisfiable, $x = 1$ can be added to t_{k+1}.

Theorem 1: Any generalized counterexample c to FGq may be extended to a concrete counterexample $\tilde{c}$.

Proof: Consider a generalized counterexample c to FGq with its lasso state repeating at times m and $n > m$. By the discussion above, we can find a concrete trace $\tilde{c}$ which agrees with valuations in c, though the states at times m and n in $\tilde{c}$ may not be identical. However, since

Algorithm 1 Trace Concretization

Input: A trace $\langle t_0, \ldots, t_n \rangle$
Output: A trace $\langle \tilde{t}_0, \ldots, \tilde{t}_n \rangle$ with $t_k \implies \tilde{t}_k$ for all k.

1: $\tilde{t}_0 \leftarrow ConcretizeInitial(t_0)$
2: **for** $k = 0, \ldots, n-1$ **do**
3: $\quad \tilde{t}_{k+1} \leftarrow ConcretizeForward(\tilde{t}_k, t_{k+1})$

Algorithm 2 Trace Generalization

Input: A trace $\langle t_0, \ldots, t_n \rangle$
Output: A trace $\langle \tilde{t}_0, \ldots, \tilde{t}_n \rangle$ with $\tilde{t}_k \implies t_k$ for all k.

1: $\tilde{t}_n \leftarrow GeneralizeFinal(t_n)$
2: **for** $k = n-1, \ldots, 0$ **do**
3: $\quad \tilde{t}_k \leftarrow GeneralizeBackward(t_k, \tilde{t}_{k+1})$

the loop of c can be infinitely unrolled, assuming a finite system, eventually a state in $\tilde{c}$ will repeat, thus yielding a concrete counterexample. ■

Theorem 1 demonstrates that a generalized liveness counterexample may be mapped to a concrete one using simulation, implying a scalable algorithm.

B. Trace Generalization

Given a trace, we may use Algorithm 2 to generalize it. $GeneralizeFinal(t_n)$ returns a generalization of t_n. For example, if the trace witnesses a number of failures of q and $t_n \implies \neg q$, this corresponds to removing some of the other variables from t_n. $GeneralizeBackward(t_k, \tilde{t}_{k+1})$ returns a generalization of t_k which still forms a predecessor of $\tilde{t}_{k+1}$. If t_k is a propagating predecessor of t_{k+1}, then we can generalize t_k using ternary simulation: if replacing the value of a state variable in t_k by X does not influence any of the variables in t'_{k+1}, then this variable can be removed from t_k. More generally, when t_k is an implying predecessor of t_{k+1} with respect to some i_k, we can consider the unsatisfiability of $t_k \wedge i_k \wedge T \wedge \neg t'_{k+1}$ and generalize from t_k variables unnecessary in the unsatisfiable core returned by the SAT solver.

C. Modifying Traces with Tentative Loops

The following example demonstrates that the processes of concretizing and generalizing a trace are both capable of creating or destroying the validity of that trace as a counterexample.

Example 5: Let q, x, y be state variables with initial values $q_0 = 1$, $x_0 = 0$, $y_0 = 0$ and next-state values $q' = q \wedge x$, $x' = x$, $y' = \neg y$. The concrete trace $(1,0,0) \rightarrow (0,0,1) \rightarrow (0,0,0)$ does not exhibit a counterexample to FGq. A partially-generalized trace $(1,0,\cdot) \rightarrow (0,0,\cdot) \rightarrow (0,0,\cdot)$ does exhibit a counterexample. A futher-generalized trace $(1,0,\cdot) \rightarrow (0,0,\cdot) \rightarrow (0,\cdot,\cdot)$ again does not exhibit a counterexample.

Design	k generalized	k concrete	k modified
cubak	20	20	20
cujc128f	5	1	1
cutf2	9	12	5
cutq2	16	16	12
lmcs06dme2p0	4	5	4

TABLE I
VALUES OF k YIELDING VALID COUNTEREXAMPLES

Design	k-LIVENESS	BMC
cubak	295s	12084s
cuhanoi10	5s	3492s

TABLE II
k-LIVENESS WITH INTERNAL IC3 TRACE VS. BMC

Consider a generalized trace $\langle t_0, \ldots, t_n \rangle$ with a pair of indices $i < j$ such that $t_i \wedge t_j \neq \bot$ and $\exists k \in [i..j]$ with $t_k \implies \neg q$. The condition $t_i \wedge t_j \neq \bot$ means that there is no state variable present in opposite polarities in t_i vs t_j. We call $\langle t_i, \ldots, t_j \rangle$ a *tentative* loop. We propose the following technique, referred to as $ConcretizeTentative(i,j)$: starting from t_i, concretize the trace forward by conjuncting the states $t_{i+1}, \ldots, t_j$ with the values forced by propagation. In this way, the concretized state t_j might now become more concrete than t_i yielding a counterexample. One may further tailor the concretization process to yield a repeating state when possible via an appropriate SAT query.

IV. COUNTEREXAMPLES VIA k-LIVENESS

The k-LIVENESS algorithm of [1] proves properties of form FGq by bounding the number of times that q can become false: if there are no traces with more than k occurrences of $\neg q$, then on every trace q must eventually become true forever. The algorithm works by gradually increasing k until a proof is obtained.

When FGq does not hold, it is noted in [1] that a bounded counterexample trace for some k may be analyzed to see if it is a valid unbounded counterexample: given a finite system and large-enough k, there must be a trace with a repeated state. Though for a realistic system, it is stipulated that k would likely need to be impractically large.

Surprisingly, we find that the opposite is true: on 44 of the HWMCC'12 benchmarks with failing liveness properties, the traces returned by the underlying safety model checker exhibit a counterexample with reasonably-small values of k. Additionally, on most of these one may detect a counterexample for even smaller values of k by manipulating traces with $ConcretizeTentative$. A few selected results are presented in Table I.

As in [1], we have implemented k-LIVENESS on top of IC3/PDR. PDR minimizes proof obligations using

ternary simulation [5], and thus directly yields generalized counterexamples for bounded property failures. Column 2 corresponds to the smallest value of k for which this generalized trace kept internally by IC3 exhibits a **generalized** counterexample. Column 3 corresponds to the smallest k for which the concretization of the trace from Column 2 using Algorithm 1 exhibits a **concrete** counterexample. The final column uses $ConcretizeTentative(i, j)$ on the trace of Column 2, for each tentative loop $\langle t_i, \ldots, t_j \rangle$ therein.

On cutf2 and lmcs06dme2p0, considering generalized traces detects counterexamples earlier due to removal of irrelevant state variables. On cujc128f, removing state variables from later timesteps precludes the detection of counterexamples. And on cutf2, partial concretization of the generalized trace yields a counterexample earlier than the other two methods.

Regarding impact on verification resources: on most of the *failing liveness* HWMCC'12 testcases, direct bounded model checking (BMC) often yields a counterexample with significantly lesser resources than k-LIVENESS augmented with our techniques. We note that the set of public liveness testcases is unfortunately quite small. Nonetheless, our techniques in cases are substantially faster than existing method such as BMC: see Table II. This offers a some evidence of the practical utility of our techniques on classes of complex problems.

V. Searching for Generalized Counterexamples

In this section we present an algorithm which directly searches for a minimal *propagating* generalized counterexample. This algorithm uses bounded model checking applied to a ternary-valued encoding of the netlist. This algorithm incrementally increases the unfolding depth n every time it proves that no generalized counterexample of length $\leq n$ exists.

For a given n, we seek a sequence $t_0, \ldots, t_n$ of generalized states and a sequence $i_0, \ldots, i_n$ of inputs so that the following conditions are satisfied:

1) t_0 contains an initial state;
2) for each $k \in [0..n-1]$ the assignments to t_k and to i_k alone imply t_{k+1};
3) $\exists m \in [0..n-1]$ such that $t_m \implies t_n$;
4) $\exists k \in [m..n-1]$ such that $t_k \implies \neg q$.

Note that every concrete lasso-shaped counterexample satisfies these conditions, thus if there are concrete counterexamples of length n, the suggested scheme will succeed with the value n or less.

Unfortunately, on the limited set of failing HWMCC'12 benchmarks, the minimal length of a propagating counterexample is the same as the minimal length of a concrete counterexample, and so the proposed scheme does not help. On the other hand, on contrived Examples 1-3, this algorithm detects generalized counterexamples of length 2 for any size of cnt, which not surprisingly may outperform by a large degree other techniques which search for a concrete counterexample.

VI. Related Work

The concept of minimizing counterexample traces has been explored extensively for a variety of purposes such as enhanced debugging, e.g. [2]. A related concept of generalizing a predecessor of a given state either by ternary simulation, via a SAT solver, or using quantifier elimination has also been widely explored, e.g. [6]. However, a significant distinction is that we we consider generalized counterexamples to *liveness* properties which can be significantly shorter than concrete counterexamples, and as such dedicated algorithms which search for *generalized* counterexamples may be developed.

The work of [4] addresses the topic of netlist transformations which preserve the existence of a liveness counterexample. For example, the cone-of-influence reduction combined with other netlist rewriting techniques can remove various signals from the netlist, thus possibly shortening lengths of counterexamples. However, netlist transformations apply to *all* time-frames and all possible traces, which does not offer the granularity of state-specific reductions enabled by our technique.

Cycle-dependent abstractions do allow the granularity of abstracting variables irrelevant *at a particular timestep*, though are typically only applicable as embedded in specific proof techniques (e.g., [7]). However, in general the existence of a counterexample on an abstracted model does not imply the existence of a counterexample on the concrete model. Additionally, this prior work does not address shortening of liveness counterexamples.

References

[1] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, 2012.
[2] K.-H. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD*, 2005.
[3] Dong Wang et al., "Formal property verification by abstraction refinement with formal, simulation and hybrid engines," in *DAC*, 2001.
[4] J. Baumgartner and H. Mony, "Scalable liveness checking via property-preserving transformations," in *DATE*, 2009.
[5] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011.
[6] P. Chauhan, E. M. Clarke, and D. Kroening, "Using SAT based image computation for reachability analysis," 2003.
[7] L. Zhang, M. Prasad, and M. Hsiao, "Interleaved invariant checking with dynamic abstraction," in *CHARME*, 2005.

The Design and Implementation of the Model Constructing Satisfiability Calculus

Dejan Jovanović
New York University

Clark Barrett
New York University

Leonardo de Moura
Microsoft Research

***Abstract*—We present the design and implementation of the Model Constructing Satisfiability (MCSat) calculus. The MCSat calculus generalizes ideas found in CDCL-style propositional SAT solvers to SMT solvers, and provides a common framework where recent model-based procedures and techniques can be justified and combined. We describe how to incorporate support for linear real arithmetic and uninterpreted function symbols in the calculus. We report encouraging experimental results, where MCSat performs competitive with the state-of-the art SMT solvers without using pre-processing techniques and ad-hoc optimizations. The implementation is flexible, additional plugins can be easily added, and the code is freely available.**

I. Introduction

Considering the theoretical hardness of SAT, the astonishing adeptness of SAT solvers when attacking practical problems has changed the way we perceive the limits of algorithmic reasoning. Modern SAT solvers are based on the idea of *conflict-driven clause learning* (CDCL) [1]–[3]. The CDCL algorithm is a combination of an explicit backtracking search for a satisfying assignment complemented with a deduction system based on Boolean resolution. In this combination, the worst-case complexity of both components is often circumvented by the components guiding and focusing each other.

Generalization of the SAT problem to the first-order domain is called satisfiability modulo theories (SMT). On the shoulders of efficient SAT solvers, and numerous successful applications, SMT has gained momentum as a more expressive and equally performant framework. The common approach to solving SMT problems is to employ a SAT solver to enumerate assignments of the Boolean abstraction of the formula. The candidate (partial) Boolean assignments are then either confirmed or refuted by a *decision procedure* dedicated to reasoning about conjunctions of theory-specific constraints. If multiple theories are involved, satisfiability in the combination of such theories can be ensured by relying on high level combination frameworks in the spirit of Nelson and Oppen [4]–[6]. This style of reasoning is commonly called DPLL(T) [7], [8] and is employed by most of the SMT solvers today.

The Model-Constructing Satisfiability (MCSat) calculus [9] provides a more general alternative to DPLL(T), lifting the idea of the CDCL-style model construction with conflict resolution to the first-order domain. MCSat encompasses many recent model-based decision procedures for theories such as linear real arithmetic [10]–[12], linear integer arithmetic [13], nonlinear arithmetic [14], and floating-point [15] arithmetic.

Although the model-based decision procedures have proved effective for theories of high complexity, it was unclear whether the approach could be used with combinations of theories, and whether the approach could be competitive for "simple" theories where incumbent solutions seem to be satisfactory. In this paper we describe an implementation of the MCSat framework that can reason effectively in the combination of linear real arithmetic and uninterpreted functions, providing positive answers to both concerns. The procedure for linear arithmetic is a careful but conceptually simple implementation of a model-driven Fourier-Motzkin elimination, while the combination with uninterpreted functions is provided through model-driven Ackermanization [5], [16].

II. Preliminaries

We assume that the reader is familiar with the usual notions and terminology of propositional and first-order logic (see e.g. [17]).

As usual, we will denote the set of rational numbers as $\mathbb{Q}$ and use a, b, c to denote constants from $\mathbb{Q}$. We assume a finite set of Boolean and real variables, denoting them with letters x, y, z, and a finite set of uninterpreted function symbols which we denote with letters f, g. Each such function symbol f is associated with a fixed arity $k > 0$. We define a UF pure term inductively, with variables and constants being UF pure terms, and a function term $f(t_1, \ldots, t_k)$ being UF pure if each t_i is a UF pure term. For example, $f(1)$ and $f(f(x))$ are UF pure, but $x + y$ and $f(x + y)$ are not. We will refer to non-constant UF pure terms as *generalized variables* and, with abuse of notation, we will also refer to them with letters x, y, z. Intuitively, generalized variables are terms seeking an interpretation.

We use p, q to denote linear polynomials over generalized variables with coefficients in $\mathbb{Q}$. All polynomials are assumed to be in sum-of-monomials normal form $a_1x_1 + \cdots + a_nx_n + c$, with a_i and c being constants, and x_i denoting generalized variables. For linear polynomials p and q, a linear constraint is a constraint of the form $p \bigtriangledown q$, where $\bigtriangledown \in \{<, \leq, =\}$.

An *atom* is either a Boolean variable or a linear constraint, and we consider atoms to be generalized variables of Boolean type. A *literal* L is an atom or a negation of an atom. A *clause* C is a disjunction of literals $(L_1 \vee \cdots \vee L_n)$, and we denote the empty clause with $\bot$.

The research reported in this paper was supported in part by NSF grant CNS-1228768.

ISBN 978-0-9835678-3-7/13.

Example II.1. *Consider the constraints*

$$f(x+1) < y,\ \ x = y\ .$$

The term $f(x+1)$ is not pure, but we can purify the constraints by introducing a new variable s_1 obtaining

$$f(s_1) < y,\ \ s_1 = x+1,\ \ x = y\ .$$

The constraints above are satisfiable, for example, by the interpretation $x \mapsto 1,\ y \mapsto 1,\ s_1 \mapsto 2,\ f(s_1) \mapsto 0$.

A. Deduction Rules

MCSat as a proof system is a clausal deduction system based on clausal inference rules. Given a set of input clauses, MCsat either finds an assignment of variables that satisfies the clauses, or derives a proof of the unsatisfiability using the rules below.

The core of MCSat is driven by the *Boolean resolution* rule. Given two clauses $C \vee L$ and $\neg L \vee D$, we can eliminate the literal L using the Boolean resolution rule

$$\frac{C \vee L \qquad \neg L \vee D}{C \vee D}$$

We denote the result of applying the resolution rule with $\mathsf{resolveB}(C, D, L)$.

For reasoning in linear arithmetic we use the *Fourier-Motzkin rule*. Given two inequalities $(p_L < x)$ and $(x < p_U)$, we can eliminate the variable x using the Fourier-Motzkin rule, obtaining a new inequality $(p_L < p_U)$. In clausal form, this rule can be stated as

$$\frac{}{\neg(p_L < x) \vee \neg(x < p_U) \vee (p_L < p_U)}$$

We denote this rule with $\mathsf{resolveFM}$. The rule above is applied to strict inequalities and, as expected, we overload $\mathsf{resolveFM}$ to cover non-strict inequalities and equalities.

In addition to the Fourier-Motzkin rule, in order to reason about dis-equalities we also use the *equality split rule*, which states that the relation between polynomials p and q can only be one of the three.

$$\frac{}{(p = q) \vee (q < p) \vee (p < q)}$$

We denote the split rule with $\mathsf{splitEq}$.

For reasoning about uninterpreted functions we use the *Ackermann expansion rule* which states that, for any uninterpreted function symbol f of arity k, if $x_i = y_i$ for $i = 1, \ldots, k$, then also $f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)$, or, in clausal form

$$\frac{}{x_1 \neq y_1 \vee \cdots \vee x_k \neq y_k \vee f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)}$$

We denote the Ackermann rule with $\mathsf{resolveCC}$.

We also assume a general "normalization" rule that performs simple semantics-preserving transformations on clauses, denoted with a dashed line, such as

$$\frac{\neg(p < q) \vee (x < 0) \vee (x < 0)}{(q \leq p) \vee (x < 0)}$$

The four rules above, together with the normalization rule, comprise the whole of our proof system, which speaks to the simplicity of the MCSat approach.

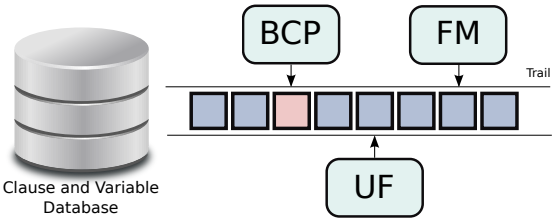


Fig. 1. Main solver components

III. Core Architecture

The MCSat architecture consists of a core solver that manages the solver components depicted in Figure 1. The main components are the clause and variable database, the solver trail, and the reasoning plugins. The core solver drives the solving process, and is responsible for dispatching notifications and handling requests from the plugins, while the plugins reason about the content of the trail. The most important duty of the core is to perform conflict analysis when the reasoning plugins detect a conflicting state.

The clause database contains a compact representation of all the clauses in the system (including unit clauses). The clause database contains both the input clauses and the clauses learned during the search. The variable database maintains the information about all the generalized variables in the system in an efficient index-based form. Both databases are occasionally compacted, by heuristically marking the clauses to keep, and then garbage-collecting unmarked clauses and variables using a mark-and-sweep approach. Maintaining a balanced amount of clauses and variables is crucial in the MCSat setting as, in contrast to the DPLL(T) framework, model-based decision procedures can generate many new literals that would, if not removed, eventually overwhelm the system.

Plugins are the standard way of extending MCSat, and we currently have implementations of dedicated plugins that can reason about the Boolean structure, linear real arithmetic, and uninterpreted function symbols, described in Section IV.

A. The Trail

The central data-structure in the framework is the solver trail. It is a generalized version of the trail found in modern SAT solvers. In our design of the interface to the trail, we make sure that the trail is used not only as the container of reasoning history, but also as the object ensuring formal progress towards termination of the search process.

The trail is a sequence of trail elements, where each element can be either a Boolean decision, a semantic decision, a clausal propagation, or a semantic propagation.

A *Boolean decision* is a literal L that we assume to be true, and is emphasized in the trail as $\mathbf{L}$. These elements are the equivalent of decided literals found in modern SAT solvers. A *semantic decision* is a decision on the value of a *non-Boolean* generalized variable. We write semantic decisions as $\mathbf{x} \mapsto \alpha$, where α represents a value from the type of the variable x. For example, we may decide that the value of a real variable

x is 1.[1] A *clausal propagation* is a literal L derived to be true through clause C using Boolean constraint propagation (BCP), which is marked as $L_{\downarrow C}$. If the clause $C = L$, i.e. it is a unit clause, we mark the propagation just as $L_{\downarrow}$. The *level* of a decision element, or a clausal propagation, is the number of decisions in the trail up to and including the element itself.

We say that a literal L (term t) *can be evaluated* if the generalized variables that appear in the literal L (term t) are all assigned values in the trail through semantic decisions. A *semantic propagation* marks a literal L that can be evaluated to **true**. We denote semantic propagation as $L_{\downarrow k}$, where the value k represents the level of the highest semantic decision used in evaluating L. The *level* of a semantic propagation $L_{\downarrow k}$ is k. If a literal L ($\neg L$) appears on the trail M as an element of level k, we say that L is **true** (**false**) in M, and the level of L is k.

Example III.1. *Consider the clause*

$$C \equiv (0 < x) \vee (0 < y) \vee (1 < x + y)$$

and the trail

$$M = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{0} \overset{2}{<} \mathbf{y}),\ \neg(0 \overset{1}{<} x)_{\downarrow 1},\ (1 < x \overset{2}{+} y)_{\downarrow C}]\!] \ .$$

The level of each element is marked above it in the trail. The first two elements of M are a semantic decision assigning the variable x to the value 0, and a Boolean decision of the literal $\neg(0 < y)$. The next element of M is a semantic propagation of $\neg(0 < x)$. Note that the semantic propagation is marked at level 1, as this is the level at which x is assigned. The last element of M is a clausal propagation of the literal $(1 < x+y)$ due to clause C.

Adding new elements to the end of the trail, by performing a decision, or propagating a literal, is restricted as follows.

- No literal L can be added to the trail (either by decision or propagation), if the literal L or $\neg L$ already appears on the trail.
- No semantic decisions $\mathbf{x} \mapsto \alpha$ can be added to the trail if it invalidates a literal L that is already on the trail, i.e. if $\neg L$ would be a semantic propagation after the decision.
- No Boolean decision $\mathbf{L}$ can be added to the trail if it invalidates a clause C in the system, i.e. if the literal $\neg L$ appears in C, and all other literals of C are already **false**.
- A clausal propagation $L_{\downarrow C}$ can only be added to the trail if the literal L appears in C, and all other literals of C are already **false**.

Besides adding elements to the trail, a trail can also be *backtracked*. Backtracking amounts to retracting some decisions and their consequences from the trail. We require that any backtracking of the trail be accompanied with a clausal *backtracking reason* that is used to maintain the invariance of progress. The reason for backtracking is always a clause C that evaluates to **false** in the trail, thus a signal that the search must be revised. In addition to being false, the clause C should also be either a *unique implication point* [2] (UIP) clause or a *semantic split* clause, as explained below, and we denote this with the predicate $\mathsf{canBacktrackWith}(M, C)$.

Let $\mathsf{topLevel}(M, C)$ denote the highest level (in M) of a literal from C, and let $\mathsf{topLiterals}(M, C)$ denote the set of literals from C at this highest level. Clause C is a UIP clause if there is only one literal L in $\mathsf{topLiterals}(M, C)$. A UIP clause C can thus be used to propagate L at the second highest level of literals in C, or level 0 if C is unit, and we denote this level as $\mathsf{uipLevel}(M, C)$. Clause C is a semantic split if each $L \in \mathsf{topLiterals}(M, C)$ is a semantic propagation $L_{\downarrow k}$ in M.

Algorithm 1: MCSAT::BACKTRACKWITH(M, C)

Data: trail M, clause C evaluates to **false** in M
if *C is a UIP clause in M* **then**
  $level \leftarrow \mathsf{uipLevel}(M, C)$
else C is a semantic split clause in M
  $level \leftarrow \mathsf{topLevel}(M, C)$ - 1
remove from M all elements of level $> level$
if *C was a UIP clause* **then**
  for the unassigned $L \in C$, add $L_{\downarrow C}$ to M
else C was a semantic split clause
  for an unassigned $L \in C$, add decision $\mathbf{L}$ to M

The backtracking procedure `backtrackWith` is presented in Algorithm 1. We call propagations, decisions, and backtracking *valid* if they conform to the restrictions outlined above.

Example III.2. *Consider again the clause C and trail M from Example III.1. Using the Fourier-Motzkin rule, we can deduce the following valid clause*

$$\frac{\neg(1 - x < y) \vee \neg(y \leq 0) \vee (1 - x < 0)}{R_1 \equiv \neg(1 < x + y) \vee (0 < y) \vee (1 < x)}$$

The first two of the literals from R_1 are already **false** *in the trail M, and the last literal $(1 < x)$ is a new literal. The new literal can be evaluated in M, and we can propagate it semantically, obtaining a new trail*

$$[\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0}, \neg(\mathbf{0} \overset{2}{<} \mathbf{y}), \neg(0 \overset{1}{<} x)_{\downarrow 1}, (1 < x \overset{2}{+} y)_{\downarrow C}, \neg(1 \overset{1}{<} x)_{\downarrow 1}]\!] \ .$$

The clause R_1 is valid and evaluates to **false** *at M, i.e. the search needs to be revised. But, R_1 is not suitable for backtracking since it contains two literals of level 2, and none of them is a semantic propagation. Fortunately, we can use the Boolean resolution rule to remove the literal propagated by C and deduce*

$$\frac{C \qquad R_1}{R_2 \equiv (1 < x) \vee (0 < x) \vee (0 < y)}$$

The clause R_2 only contains one literal at the highest level, so it is an UIP clause suitable for backtracking. The effect of backtracking the trail with `backtrackWith(`R`)` *is a new*

[1] A Boolean decision is in fact just a special case of a semantic decision, but we consider them separately for presentation and implementation purposes.

trail

$$[\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0}, \neg(0 \overset{1}{<} x)_{\downarrow 1}, \neg(1 \overset{1}{<} x)_{\downarrow 1}, (0 \overset{1}{<} y)_{\downarrow R_2}]\!] \ .$$

Note that, in addition to the propagated UIP literal, the backtracking procedure also kept the late semantic propagations.

It is important to note that we diverge slightly from the original MCSat presentation [9], in that the trail as presented here contains explicit semantic propagation elements. In [9], all semantic propagation was implicit in the trail and available through evaluation. This change was guided by implementation reasons since it allows, for example, a more efficient implementation of Boolean constraint propagation. The change is a minor one, and the main theoretical results from [9] hold in this setting and we restate the most important ones as lemmas.

It was shown in [9] (Theorem 1) that using valid trail operations is enough to ensure termination, if we assume that all literals that the system will ever see come from a finite set of literals $\mathcal{B}$, that we call the *finite basis*.

Lemma III.1. *Starting from an empty trail $[\![]\!]$, any procedure that only uses valid trail operations, while using only literals from the finite basis $\mathcal{B}$, can only make a finite number of such trail operations.*

In Section IV we will show how the finite basis assumption can be ensured for problems combining linear arithmetic and uninterpreted functions.

B. Conflict Analysis

As in CDCL SAT solvers, conflict analysis is used to learn from the conflicting clauses encountered during the search – clauses with all literals **false** in the trail. As seen in Example III.2, it is possible to identify clausal conflicts that can not directly be used for backtracking. Conflict analysis takes a conflicting clause and transforms it into a new clause that is suitable for backtracking. This newly learned clause is used in the main solver loop to revise the search.

A conflicting clause is not suitable for backtracking if it contains more than one literal at the top level, and these literals are not semantic propagations. It is easy to show that the problematic literals are clausal propagations, and conflict analysis can eliminate them by performing Boolean resolution with the clauses that propagated them. Since the analysis uses existing valid clauses and the resolution rule, the result of such conflict analysis will be a valid deduction. Therefore, if conflict analysis learns an empty clause $\bot$, this will be a signal to the main solver that the problem is unsatisfiable. The conflict analysis procedure is presented in Algorithm 2.

Note that the analysis procedure only uses the resolveB rule. Other deduction rules are "axiom" rules used to create new clauses which can be used to identify conflicting situations or for propagation purposes. Also, as the analysis procedure concludes as soon it finds a clause suitable for backtracking, if this is the case due to a UIP clause, the analysis style corresponds to the 1st UIP SAT strategy [18].

Algorithm 2: MCSAT::ANALYZECONFLICT(M, C)

Data: solver trail M, clause C inconsistent with M

```
k ← M.size()
while C ≠ ⊥ and ¬ canBacktrackWith(M, C) do
    k ← k − 1
    if M[k] = L_↓D and ¬L ∈ C then
        C ← resolveB(C, D, L)
return C
```

Lemma III.2. *Given a valid trail M and a clause C that is* **false** *in M, the* `analyzeConflict(`M, C`)` *procedure always terminates with a clause R that is a valid deduction and is either the empty clause or is suitable for backtracking.*

C. Main Search Loop

The algorithm behind MCSat is based on the search-and-resolve loop common in modern SAT solvers (e.g. [19]). The main loop of the solver performs a "smart" search for a satisfying assignment and terminates either by finding the assignment that satisfies the original problem, or deduces that the problem is unsatisfiable. The main `check()` method of the solver is presented in Algorithm 3.

Algorithm 3: MCSAT::CHECK()

```
1  while true do
2      propagate()
3      if detected conflict with clause C then
4          R ← analyzeConflict(M, C)
5          if R = ⊥ then return unsat
6          backtrackWith (M, R)
7      else
8          if queue.empty() then return sat
9          x ← queue.pop()
10         decideValue(x)
```

The search process goes forward, making continuous progress, either through propagation, conflict analysis, or by making a decision. The `propagate()` procedure invokes the propagation procedures provided by the enabled plugins. Each plugin is allowed to propagate new information to the top of the trail. If a plugin detects an inconsistency this is communicated to the solver by producing a conflicting clause. This is recorded by the solver and allows the solver to analyze the conflict using the `analyzeConflict()` procedure. If conflict analysis learns the empty clause $\bot$, the problem is proved unsatisfiable, otherwise the learned clause is used to backtrack the search.

On the other hand, if the plugins have performed propagation to exhaustion, and no conflict was detected, the procedure makes progress by deciding a value for an unassigned variable. The solver picks an unassigned variable x to be assigned, and

relegates the choice of the value to the plugin responsible for assigning x. A choice of value for the selected unassigned variable should exist, as otherwise a plugin should have detected the inconsistency. MCSat uses a uniform heuristic to select the next variable, regardless of their type. The heuristic is based on how often a variable is used in conflict resolution, and is popularly used in CDCL-style SAT solvers [3]. Note that, as explained in the preliminaries, every atom (e.g., $x < 2$) is treated as a generalized Boolean variable. If all the variables are assigned to a value, this is a satisfying assignment for the original problem.

IV. Plugins

The reasoning engines in MCSat are organized in modules that we call *plugins*. The plugins can register listeners for notification about important events in the system, such as new assertion formulae, creation of new clauses and generalized variables, and garbage-collection events. Plugins participate in the solving process by performing propagation and detecting conflicts, with dedicated plugins also taking part in selecting values for variables.

In order to ensure completeness in the system, if a plugin is dedicated to selecting values for a particular type T (such as Boolean or real), it must be unit-constraint complete. We call a plugin *unit-constraint complete* for type T if, after a call to `propagate()`, either the plugin has identified a conflicting clause C, or, for each unassigned variable x of type T there exists a valid decision $\mathbf{x} \mapsto \alpha$ (or a Boolean decision if T is Boolean). Note that unit-constraint completeness *does not* require that the plugin ensures consistency of all assertions, only that the assertions with a single unassigned variable are satisfiable – a much easier property to check.[2]

Example IV.1. *Consider the clauses*

$$C_1 \equiv ((x + y \leq 0) \vee (0 \leq y) \vee z)$$
$$C_2 \equiv ((x + y \leq 0) \vee (0 \leq y) \vee \neg z) \ ,$$

where variables x and y are of real type, and the variable z is a Boolean, with the corresponding trail

$$M = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}) \ ]\!] \ .$$

In the trail M, the clauses C_1 and C_2 have all but one literal **false***, i.e. they are unit constraints that can propagate a literal. M does not allow a value for variable z, since assigning z to* **true** *invalidates clause C_1, and assigning z to* **false** *invalidates C_2. This kind of unit constraint conflict can be detected with exhaustive Boolean constraint propagation. For example, using C_1, we can propagate z obtaining*

$$M' = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}),\ z^{3}_{\downarrow C_1} \ ]\!] \ .$$

In the trail M' the clause C_2 is **false** *and is a conflicting clause.*

[2]This is a variant of local consistency closely related to forward checking [20].

In addition to the Boolean conflict above, the original trail M does not allow a selection of value for the real variable y. With respect to y there are two unit constraints in M – the constraint $\neg(x + y \leq 0) \equiv (0 < x + y)$ (that evaluates to $0 < y$) and the constraint $\neg(y \geq 0) \equiv y < 0$ – and they are in conflict.

The conflicting unit constraints can be resolved using the resolveFM *rule obtaining the clause*

$$\frac{\neg(-x \leq y) \vee \neg(y < 0) \vee (-x < 0)}{R \equiv (x + y \leq 0) \vee (0 \leq y) \vee (0 < x)}$$

We can semantically propagate that the new literal $(0 < x)$ is **false***, obtaining a trail*

$$M'' = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}),\ \neg(0 \overset{1}{<} x)_{\downarrow 1} \ ]\!] \ .$$

In the trail M'' the clause R is **false** *and is a conflicting clause.*

A. BCP And Watchlists

As hinted in Example IV.1, in order to ensure unit-completeness for the Boolean variables in a clausal setting, it is enough to perform Boolean constraint propagation (BCP) to exhaustion. We've implemented the customary efficient BCP loop in a dedicated BCP plugin, with the basic mechanics built upon the important concept of watchlists.

In the SAT literature, the two-literal watchlist was first introduced in [3] as an efficient mechanism to detect when a clause becomes unit. In the MCSat approach, the concept of watchlists is more generally applicable and we use it in other plugins too. The key insight is the following. If we are interested in detecting when, of a set V of variables, exactly k variables are left unassigned, it is enough to "watch" a set $W \subseteq V$ of $(k + 1)$ variables by maintaining the invariant that all variables in W are unassigned. If a variable $x \in W$ becomes assigned, then we try to replace x with another unassigned variable $y \in V \setminus W$. If we can't find an unassigned variable y to replace x, this means that in V there are now exactly k variables unassigned.

B. Linear Real Arithmetic

The plugin for reasoning about linear arithmetic (FM plugin) is responsible for reasoning about linear constraints and deciding values for variables of real type. In order to maintain unit completeness for variables of the real type, we should ensure that for each real variable x, the set of all linear constraints from the trail M, that are unit in variable x, is consistent.

We call a literal $L \in M$ a linear constraint unit in x, if the atom of L is a linear constraint, and all variables of L different from x are assigned in M. Any linear constraint $L \in M$, unit in x, can be equivalently written as one of

$$x \neq p, \qquad p \triangledown x, \qquad x \triangledown p \ ,$$

with $\triangledown \in \{<, \leq, =\}$. Since the constraint is unit, the polynomial p can be evaluated in M and takes some value $v \in \mathbb{Q}$.

Example IV.2. *Consider the trail*

$$M = [\![\neg(x + y < 0),\ \mathbf{x}\mapsto\mathbf{0}, \neg(x + z = 1), (0 < y + z)]\!] \ .$$

The trail M contains two unit linear constraints. The literal $\neg(x+y<0)$ is unit in variable y, is equivalent to $(-x \leq y)$, and evaluates to $0 \leq y$. The literal $\neg(x+z=1)$ is unit in z, is equivalent to $z \neq 1-x$, and evaluates to $z \neq 1$. The linear constraint $(0 < y + z)$ is not unit in M.

Using the watchlist mechanism, we can efficiently maintain an up-to-date set of linear constraints that are unit. The unit constraints in the trail impose constraints on the unit variables, and for each variable x, the FM plugin tracks the following

- the strongest lower bound of x implied by a unit linear constraint $L \in M$;
- the strongest upper bound of x implied by a unit linear constraint $L \in M$;
- a set of values v_D such that x is implied to be different from v_D by a unit linear constraint $L \in M$.

Having the information above, for each variable x, we can now effectively reason about its unit feasibility by inspecting if there is a value within its upper and lower bound that is not disallowed by a disequality constraint. If, for some variable x, there is no such value, it must be due to a *bound conflict* or a *disequality conflict*.

Variable x is in a bound conflict if the trail contains two unit linear constraints $L_L \equiv (p_L \ \triangledown_L \ x)$ and $L_U \equiv (x \ \triangledown_U \ p_U)$, with p_L and p_U evaluating to v_L and v_U, where either $v_L > v_U$, or $v_L = v_U$ but at least one of the bounds $\triangledown_L$ or $\triangledown_U$ is strict ($<$). This conflict can be resolved using the resolveFM rule

$$\frac{}{\neg(p_L \ \triangledown_L \ x) \vee \neg(x \ \triangledown_U \ p_U) \vee (p_L \ \triangledown \ p_U)}$$

where $\triangledown$ is the result of combining $\triangledown_L$ and $\triangledown_U$. This clause can be used as an explanation of the conflict since the first two literals evaluate to **false**, and the last literal doesn't contain x and can be semantically propagated as **false**.

Variable x is in a disequality conflict if the trail contains a unit disequality constraint $(x \neq p_D)$, and two unit linear constraints $(p_L \leq x)$ and $(x \leq p_U)$, with p_D, p_L and p_U all evaluating to the same value v. This conflict can be resolved using a derived rule we call resolveDiseq with the derivation presented in Figure 2. This rule produces a clause that can be used as an explanation of the conflict since the first three literals evaluate to **false** and the last two literals can be semantically propagated as **false**. The resolveDiseq rule is applicable for disequality conflicts with unit equality constraints too, and although a more precise rule exists, we use this one for simplicity.

In addition to detecting conflicts, the FM plugin also eagerly performs semantic propagation. Using the same watch-list mechanism, the FM plugin tracks all linear constraints in the system, and can detect when a linear constraint becomes fully assigned. Such constraints are evaluated using the assignment in the trail and added to the trail as semantic propagations.

Computing bounds implied by unit constraints and performing semantic propagation of fully assigned linear constraints can be very expensive. The propagation loop of the FM plugin spends 90% or more of its time evaluating these constraints. In order to improve performance we use the *value time-stamping* feature of the main solver. The main solver maintains a global ever-increasing time-stamp for decision values. Each variable x is associated with its own time-stamp, and the time-stamp of x gets assigned to the global time-stamp every time x is assigned to a value different from the value that x was assigned to in the previous attempt. This allows us to detect when a set of variables (say variables of a linear constraints) are assigned to the same values as the previous time we considered the set, by keeping the maximal time-stamp of those variables. This in turn allows us to cache bound computations and semantic evaluations of linear constraints, in cases when the same values were chosen.

If no conflicts were detected, the FM plugin is dedicated to picking the values of the real variables. In order to improve performance of arithmetic operations, when deciding on a value for a variable, if possible, we always choose the values to be dyadic rationals.[3] Additionally, if allowed by the bounds, when picking a value for a variable x, we try to use the value that was used for x previously (value-caching). This is a strategy similar to phase-caching in SAT solvers [21], and allows for better evaluation cache performance when using value time-stamping described above.

C. Uninterpreted Functions

Most decision procedures for uninterpreted functions are based on fast union-find algorithms complemented with congruence-closure reasoning [22], [23]. Instead, we adopt a very simple approach to reasoning about uninterpreted functions that detects direct conflicts in term assignments.

We say that a function term $f(x_1, \ldots, x_n)$ has an *evaluation representative* $f(\alpha_1, \ldots, \alpha_n)$ in a trail M, if each x_i is either the constant α_i, or is assigned by M to value α_i. For each uninterpreted function term that appears in the input formula (generalized variables), we maintain a single-variable watchlist of its non-constant arguments. This allows us to detect when all of the arguments of the function application have been assigned, and the term therefore has an evaluation representative. The UF plugin can then detect a conflict if two terms with the same evaluation representative are ever assigned to different values, and then explain the conflict using the resolveCC rule.

Example IV.3. *Consider the unit constraint $f(x) < f(y)$ and assume that the trail is in the state*

$$M = [\![(f(x) \overset{0}{<} f(y))_{\downarrow},\ \mathbf{f(x)}\overset{1}{\mapsto}\mathbf{0},\ \mathbf{f(y)}\overset{2}{\mapsto}\mathbf{1},\ \mathbf{x}\overset{3}{\mapsto}\mathbf{0}]\!] \ .$$

In this state, the UF plugin knows that the arguments of $f(x)$ are fully assigned, with the evaluation representative $f(0)$, and is assigned to 0.

[3]Dyadic rationals $\mathbb{D} = \{\frac{p}{2^k} \mid p \in \mathbb{Z},\ k \in \mathbb{N}\}$ are a convenient dense sub-ring of $\mathbb{Q}$, allowing more efficient ring operations ($+$, $\times$) due to less gcd computation.

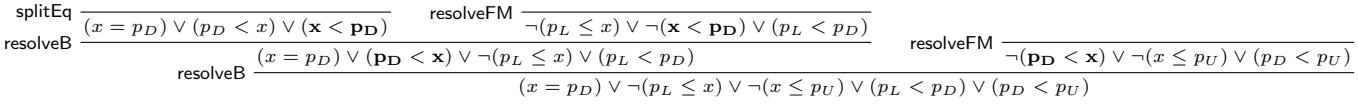


Fig. 2. Derivation of the disequality lemma.

We continue from this state to assign the next unassigned variable y, and the responsible plugin (FM) can assign it to any value, including

$$[\![M, \mathbf{y} \overset{4}{\mapsto} \mathbf{0}]\!] \ .$$

The UF Plugin now has enough information to detect a conflicting state: the term $f(y)$ has all arguments assigned, with the representative $f(0)$ that is already assigned to the value $0 \neq 1$. We can explain the conflict using the resolveCC *rule to obtain the explanation clause*

$$\frac{}{R \equiv \neg(x = y) \vee (f(x) = f(y))}$$

We can propagate the new literals semantically, adding $(x = y)_{\downarrow 4}$ and $\neg(f(x) = f(y))_{\downarrow 2}$ to the trail and marking a conflict with the clause R. The single top literal of R being **false** *makes this clause an UIP clause and the solver can then backtrack to resolve the conflict, obtaining*

$$[\![M, \neg(f(x) \overset{2}{=} f(y))_{\downarrow 2}, \ \neg(x \overset{3}{=} y)_{\downarrow R}]\!] \ .$$

With the new trail, the FM plugin can now make a more informed decision on the value of y, which will satisfy the constraints.

D. Finite Basis

In order to guarantee termination through Lemma III.1, we need to guarantee that starting from the initial problem, the literals that the procedure operates on can be bound to a finite set. New literals are only created by the plugins, particularly the FM and UF plugins, as part of clauses that explain conflicting situations. The UF plugin only creates new literals using the resolveCC rule, introducing new equalities over function terms and their arguments. Given that the number of function terms in the input problem is finite, and no new function terms are ever introduced, the number of new literals that the UF plugin can introduce is finite. As already shown in [10]–[12], fixing the decision order on variables of real type ensures that the number of new literals introduced by the FM plugin is also finite. The argument follows from the fact that the FM rule always introduces linear constraints from existing ones, with the top variable eliminated. In practice, however, we do not enforce a fixed variable order, as the flexibility in deciding variables is crucial for performance.

V. Experimental Results

We implemented the MCSat framework as an independent engine in the CVC4 [24] solver (reusing the basic infrastructure and the parser) with the code freely available, and we refer to this implementation as mcsat.[4]

In order to evaluate the new approach, we compared our implementation with several SMT solvers that support linear arithmetic and uninterpreted functions, namely cvc4 1.2 [24], z3 4.3.1 [25], mathsat 5.1.12 [26], and yices 1.0.38 [27]. All of these solvers are DPLL(T) based and implement a variant of the simplex algorithm described in [28]. All experiments were preformed on AMD Opteron 250 2.4GHz machines with a timelimit of 30 minutes and memory limited to 2GB.

We first compared the solvers on a set of pure arithmetic benchmarks from the QF_LRA category of the SMTLIB library.[5] The results are presented in Table I and show that the new mcsat implementation is competitive with the other solvers, and even excels on some problems that are hard for the DPLL(T)-based simplex solvers (such as the clocksynchro examples).

We then evaluated the solvers on the benchmarks that combine linear arithmetic and uninterpreted functions. For this we combined the benchmarks from the QF_UFLRA and the QF_UFLIA categories of the STMLIB library, while changing all the integer problems into their real-relaxation counterpart.[6] The results are presented in Table II. The results on this set show a very robust performance of mcsat, with our implementation solving all problems, in the least amount of total time. Again, there is a category of problems (wisas) hard for the DPLL(T)-based solvers where mcsat excels.

VI. Conclusion

We presented the design and implementation of the model-based satisfiability calculus. The new solver can effectively reason in linear real arithmetic and uninterpreted functions, and is competitive with existing solvers. We proposed a simple combination mechanism for uninterpreted functions, based on model filtering, that has proven to be competitive with more sophisticated theory-combination frameworks.

We see many exciting directions for future work. In addition to integrating and developing further the existing model-based decision procedures for integer and non-linear real arithmetic, we plan to develop a decision procedure for the theory of arrays based on [29]. We also plan to work on implementing theory propagation algorithms that have proved effective in the DPLL(T) framework, and to work on integration of existing decision procedures (such as simplex) into the MCSat framework.

[4]Source code of the revision used in experiments is available at `https://github.com/dddejan/CVC4/tree/mcsat-fmcad2013` in the `src/mcsat` directory. Use with `cvc4 --enable-mcsat`.

[5]Available at `http://www.smt-lib.org/`.

[6]`sed -e s/Int/Real/g -e s/QF_UFLIA/QF_UFLRA/g`

TABLE I
COMPARISON OF MCSAT WITH OTHER SOLVERS ON QF_LRA BENCHMARKS.

	mcsat		cvc4		z3		mathsat5		yices	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
clocksynchro (36)	**36**	**123.11**	36	1166.55	36	1828.74	36	1732.59	36	1093.80
DTPScheduling (91)	**91**	**31.33**	91	72.92	91	100.55	89	1980.96	91	926.22
miplib (42)	8	97.16	**27**	**3359.40**	23	3307.92	19	5447.46	23	466.44
sal (107)	107	12.68	107	13.46	107	6.37	107	7.99	**107**	**2.45**
sc (144)	144	1655.06	144	1389.72	144	954.42	144	880.27	**144**	**401.64**
spiderbenchmarks (42)	42	2.38	42	2.47	42	1.66	42	1.22	**42**	**0.44**
TM (25)	25	1125.21	25	82.12	**25**	**51.64**	25	1142.98	25	55.32
ttastartup (72)	70	4443.72	72	1305.93	72	1647.94	72	2607.49	**72**	**1218.68**
uart (73)	73	5244.70	73	1439.89	73	1379.90	73	1481.86	**73**	**679.54**
	596	12735.35	**617**	**8832.46**	613	9279.14	607	15282.82	613	4844.53

TABLE II
COMPARISON OF MCSAT WITH OTHER SOLVERS ON QF_UFLRA BENCHMARKS.

	mcsat		cvc4		z3		mathsat5		yices	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
EufLaArithmetic (33)	33	39.57	33	49.11	**33**	**2.53**	33	20.18	33	4.61
Hash (198)	198	34.81	198	10.60	198	7.18	198	1330.88	**198**	**2.64**
RandomCoupled (400)	400	68.04	400	35.90	400	31.44	**400**	**18.56**	384	39903.78
RandomDecoupled (500)	500	34.95	500	40.63	500	30.98	**500**	**21.86**	500	3863.79
Wisa (223)	223	9.18	223	87.35	223	10.80	223	65.27	**223**	**2.80**
wisas (108)	**108**	**40.17**	108	5221.37	108	443.36	106	1737.41	108	736.98
	1462	**226.72**	1462	5444.96	1462	526.29	1460	3194.16	1446	44514.60

REFERENCES

[1] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[2] J. P. M. Silva and K. A. Sakallah, "GRASP – a new search algorithm for satisfiability," in *ICCAD*, 1997.

[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Design Automation Conference*, 2001, pp. 530–535.

[4] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, pp. 245–257, 1979.

[5] L. de Moura and N. Bjørner, "Model-based Theory Combination," in *Satisfiability Modulo Theories*, ser. ENTCS, vol. 198, 2008, pp. 37–49.

[6] D. Jovanović and C. Barrett, "Being careful about theory combination," *Formal Methods in System Design*, pp. 1–24, 2012.

[7] S. Krstić and A. Goel, "Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL," *Frontiers of Combining Systems*, pp. 1–27, 2007.

[8] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[9] L. de Moura and D. Jovanović, "A Model-Constructing Satisfiability Calculus," in *Verification, Model Checking, and Abstract Interpretation*, vol. 7737, 2013, pp. 1–12.

[10] S. Cotton, "Natural domain SMT: A preliminary assessment," in *FORMATS*, 2010.

[11] K. L. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to richer logics," in *Computer Aided Verification*, 2009, pp. 462–476.

[12] K. Korovin, N. Tsiskaridze, and A. Voronkov, "Conflict resolution," *Principles and Practice of Constraint Programming*, pp. 509–523, 2009.

[13] D. Jovanović and L. de Moura, "Cutting to the chase: Solving linear integer arithmetic," in *Automated Deduction*, 2011, pp. 338–353.

[14] ——, "Solving non-linear arithmetic," *Automated Reasoning*, pp. 339–354, 2012.

[15] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *Formal Methods in Computer-Aided Design*, 2012, pp. 131–140.

[16] W. Ackermann, *Solvable cases of the decision problem*, 1954, vol. 12.

[17] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009.

[18] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Computer-aided Design*, 2001, pp. 279–285.

[19] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and applications of satisfiability testing*, 2004, pp. 502–518.

[20] C. Bessiere, "Constraint propagation," in *Handbook of Constraint Programming*, 2006, pp. 29–83.

[21] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing*, 2007, pp. 294–299.

[22] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *Journal of the ACM*, vol. 27, no. 2, pp. 356–364, 1980.

[23] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[24] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, 2011, pp. 171–177.

[25] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," *TACAS*, pp. 337–340, 2008.

[26] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS*, 2013, pp. 93–107.

[27] B. Dutertre and L. D. Moura, "The yices SMT solver," *Tool paper at* `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[28] ——, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification*, 2006, pp. 81–94.

[29] R. Brummayer and A. Biere, "Lemmas on Demand for the Extensional Theory of Arrays," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 165–201, 2009.

Trimming while Checking Clausal Proofs

Marijn J.H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler
The University of Texas at Austin

***Abstract*—Conflict-driven clause learning (CDCL) satisfiability solvers can emit more than a satisfiability result; they can also emit clausal proofs, resolution proofs, unsatisfiable cores, and Craig interpolants. Such additional results may require substantial modifications to a solver, especially if preprocessing and inprocessing techniques are used; however, CDCL solvers can easily emit clausal proofs with very low overhead. We present a new approach with an associated tool that efficiently validates clausal proofs and can distill additional results from clausal proofs. Our tool architecture makes it easy to obtain such results from any CDCL solver. Experimental evaluation shows that our tool can validate clausal proofs faster than existing tools. Additionally, the quality of the additional results, such as unsatisfiable cores, is higher when compared to modified SAT solvers.**

I. Introduction

Conflict-driven clause learning (CDCL) satisfiability solvers compute the satisfiability of a given Boolean formula. When a solver claims a formula is unsatisfiable, most solvers can also emit a proof of unsatisfiability as a sequence of learned clauses, and some solvers can produce an unsatisfiable core of the clauses used to refute a formula. Such proofs can then be checked to validate the unsatisfiability claim of a CDCL solver, while the core can be used as a starting point for extracting minimal unsatisfiable subsets (MUS) and interpolants.

Proofs of unsatisfiability can be expressed in clausal- or resolution-style formats [1], [2], [3], [4], and such proofs provide assurance that a solver is correct [5]. Any CDCL solver can emit clausal proofs with low overhead, and clausal proofs are much smaller than resolution-style proofs. However, clausal proofs are relatively expensive to validate, and clausal checkers can be complicated, making them harder to trust or mechanically verify. Although resolution proofs are easy to validate with a simple proof checker, they are hard to obtain and can be huge in size. This paper provides additional evidence that clausal proofs are more useful in practice.

SAT solvers that emit additional results [6], such as unsatisfiable cores, store an *antecedent graph*: a directed acyclic graph that represents a dependency between learned and input clauses. Storing an antecedent graph requires significant modifications to a SAT solver's implementation. The size of the antecedent graph can be two orders of magnitude larger than the size of the clause database. This requires a lot of memory, even with optimizations [7], and can slow down a solver significantly. Also, it may be hard to compute the antecedents of learned clauses for some reasoning techniques, such as equivalence reasoning and hidden literal elimination [8]. Few solvers support the storing of antecedent graphs, and no top-tier SAT solver has the ability to store them.

This paper uses clausal proof checking to produce additional results from SAT solvers. More specifically, we reconstruct an antecedent graph from a clausal proof rather than producing it while solving. Goldberg and Novikov [1] proposed an algorithm, known as *backward checking*, to achieve this. As far as we know, there is no available implementation of this method. We noticed that this method can be very expensive when used on clausal proofs from state-of-the-art CDCL solvers. In this paper, we present two optimizations for this algorithm [1] to reduce its computational costs substantially: we add clause deletion information to clausal proofs and we develop an alternative procedure to perform unit propagation.

We have implemented a proof-checking tool, called DRUP-trim, that mitigates one of the main drawbacks of clausal proof checking, namely speed. CDCL SAT solvers can easily emit clausal proofs, and these proofs can now be used to produce additional results from any SAT solver. Our DRUP-trim tool also enables validation of lookahead SAT solvers [9]. These solvers use several types of *local learning* that make it hard to emit resolution proofs; however, clausal proofs are easy to emit. Our work is most closely related to that of Van Gelder [3] whose RUP2RES tool converts clausal proofs into resolution proofs. In contrast to RUP2RES, our DRUP-trim tool can emit additional results such as unsatisfiable cores and reduced proofs. DRUP-trim does not store arcs in the antecedent graph and therefore does not suffer from high memory consumption. A slightly modified version of DRUP-trim was used to validate the unsatisfiability results of SAT Competition 2013.

Our contributions are in three areas: verification of unsatisfiability proofs, minimal unsatisfiable core extraction, and computation of Craig interpolants. Our DRUP-trim tool facilitates fast validation of unsatisfiability results of CDCL solvers and, in the process, generates additional results that can be used as a starting point for tools that produce MUSes or Craig interpolants. Most preprocessing techniques used in state-of-the-art CDCL solvers [10] can be easily converted into clausal proofs which can be used to obtain additional results for problems that are too hard to solve without them.

Our paper begins with an introduction to satisfiability, resolution, Boolean constraint propagation, and clausal proofs in Section II. In Section III, we review antecedent graphs and their applications and optimizations. Next, we present a series of improvements to clausal proof checking: backward reverse unit propagation (Section IV), the addition of clause deletion information (Section V), and a preference for clauses that are already marked as part of the core (Section VI). In Section VII, we evaluate our method and we conclude in Section VIII.

ISBN 978-0-9835678-3-7/13.

II. Preliminaries

We briefly review necessary background concepts: conjunctive normal form (CNF), resolution, Boolean constraint propagation, and clausal proofs.

A. Conjunctive Normal Form

For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a finite disjunction of literals, and a CNF *formula* is a finite conjunction of clauses. The set of literals occurring in a CNF formula F is denoted by $\mathrm{LIT}(F)$. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathrm{LIT}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. An assignment can also be thought of as a conjunction of literals. Furthermore, given an assignment τ:

- A clause C is *satisfied* by τ if $\tau(l) = \mathbf{t}$ for some $l \in C$.
- A clause C is *falsified* by τ if $\tau(l) = \mathbf{f}$ for all $l \in C$.
- A formula F is *satisfied* by τ if $\tau(C) = \mathbf{t}$ for all $C \in F$.
- A formula F is *falsified* by τ if $\tau(C) = \mathbf{f}$ for some $C \in F$.

A CNF formula with no satisfying assignments is called *unsatisfiable*. A clause C is *logically implied* by formula F if adding C to F does not change the set of satisfying assignments of F.

B. Resolution

The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2, while C_1 and C_2 are the *antecedents* of C. We write $C = C_1 \bowtie C_2$. The resolvent C is logically implied by any formula containing C_1 and C_2.

C. Boolean Constraint Propagation

A clause C is *unit* under an assignment τ if (1) there exists exactly one literal $l \in C$ such that $l \notin \tau$ and $\bar{l} \notin \tau$, and (2) for all $l' \in C$ such that $l' \neq l$, $\bar{l'} \in \tau$. We say l is the *unit literal* for *unit clause* C. Given a formula F and an assignment τ, *Boolean constraint propagation* $\mathrm{BCP}(F, \tau)$, also known as *unit propagation*, repeatedly extends τ with unit literals (for a unit clause $C \in F$ under τ) until a fixed point is achieved. If at some point during BCP τ falsifies a clause, we say that BCP *derives a conflict*.

Example 1. Given the formula $F = (a \vee b \vee c) \wedge (a \vee \bar{b})$ and the assignment $\tau = (\bar{a})$, BCP extends τ with unit literal $\bar{b}$ and then with unit literal c. As a result, $\mathrm{BCP}(F, \tau) = (\bar{a} \wedge \bar{b} \wedge c)$.

D. Clausal Proofs

Goldberg and Novikov [1] introduced *clausal proofs* as an alternative to resolution-style proofs [2] of unsatisfiability. They observed that each clause learned by CDCL conflict analysis can be validated using BCP. Learned clauses are disjunctions of literals, and the complement of a clause, written $\bar{C}$, can be interpreted as an assignment. If $\mathrm{BCP}(F, \bar{C})$ derives a conflict, then C is logically implied by F. This process is also known as *reverse unit propagation* (RUP) [3]. Learned clauses in CDCL solvers can be checked using RUP by performing the unit propagation steps in the reverse order of the search procedure; hence the name. A clausal proof, then, consists of a sequence of learned clauses that have the *RUP property*; i.e., they can be validated using RUP. A (clausal) refutation is a proof that contains the (unsatisfiable) empty clause.

In order to distinguish learned clauses from input clauses, we appeal to the notion that *lemmas* are used to construct a proof of a theorem. Here, learned clauses are lemmas which support a theorem stating that a formula is unsatisfiable. From now on, we will use the term clauses to refer to input clauses, while lemmas will refer to learned clauses.

The elegance of clausal proofs is that they can be expressed in conjunctive normal form; however, the order of lemmas in the proof is important. Clausal proofs are significantly smaller when compared to resolution proofs, and only minor modifications of a SAT solver are required to output clausal proofs. However, clausal proof checking can be quite expensive. And, checking algorithms for clausal proofs are also typically more complex than those for resolution proofs, making it harder to trust or prove correctness of the algorithm.

III. Antecedent Graphs

An *antecedent graph* is a directed acyclic graph that represents the refutation of a formula. The root nodes of an antecedent graph represent the clauses in the original formula, and internal nodes represent lemmas. A directed arc from node C_1 to node C_2 signifies the use of C_1 in the construction of C_2. In other words, C_1 is an antecedent for C_2. One of the leaf nodes in the antecedent graph is the empty clause.

Example 2. Consider the CNF formula:

$$(\bar{b} \vee c) \wedge (a \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})$$

Fig. 1 shows an antecedent graph of this formula in which clauses are shortened: $\bar{b}c$ for $(\bar{b} \vee c)$. The antecedent graph consists of four lemmas including the empty clause. For each lemma, the set of incoming arcs represents the antecedents. So the antecedents of lemma (c) are clauses $(\bar{b} \vee c)$, $(a \vee c)$, and $(\bar{a} \vee b)$.

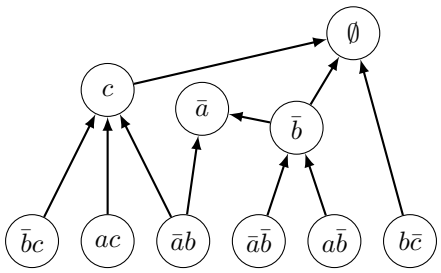


Fig. 1. Antecedent graph for an example formula. Apart from the lemma $(\bar{a})$, all clauses and lemmas are in the cone of $\emptyset$.

The *cone* of a lemma L is the set of all clauses and lemmas from which L is reachable. We refer to *core clauses* as the clauses that are in the cone of the empty clause. Similarly, *core lemmas* refer to the lemmas in the cone of the empty clause and *core arcs* are all incoming arcs of core lemmas. If a solver stores the antecedent graph, it is easy to compute the core clauses, lemmas, or arcs by simply checking for reachability to the empty clause.

The number of core arcs is typically 300 to 400 times larger than the number of core lemmas. To illustrate this difference, we computed the number of core arcs and core lemmas using Picosat [6] while solving the application benchmarks of the SAT 2009 suite, the results of which are shown as a scatter plot in Fig. 2. See Section VII for the details of the machine used in this experiment. Compared to the number of literals in core lemmas, the number of core arcs is about 10 times larger. That means that the memory consumption of a solver that stores the antecedent graph is at least 10 times larger as compared to a solver that does not store the full graph. In practice, this number can be significantly larger because the solver needs to keep some deleted lemmas — even with optimizations [7]. We observed that emitting additional results by Picosat (which requires an antecedent graph) increased the memory requirement by a factor 100 for several benchmarks. Consequently, storing the antecedent graph while solving can reduce the performance of solvers significantly and result in memory exhaustion.

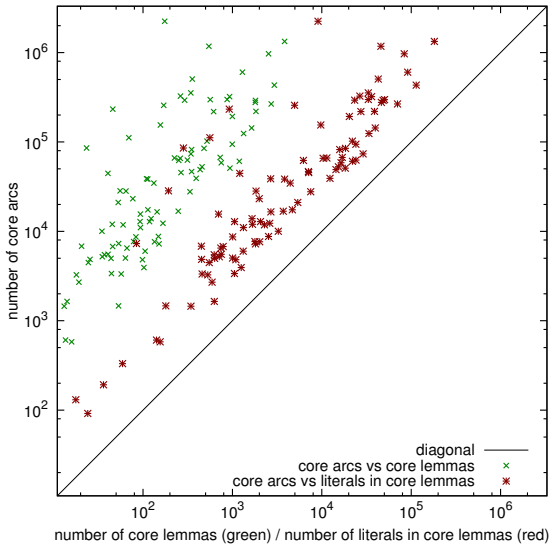


Fig. 2. A scatter plot of the number of core arcs (y-axis) versus the number of core lemmas (green) and the number of literals in core lemmas (red).

In this paper, we propose to reconstruct the antecedent graph after the search has ended by using clausal proofs. Hence, SAT solvers that want to compute additional results will no longer need to store arcs in the antecedent graph.

Applications

Verification of unsatisfiability proofs: The tool that we present in this paper is a fast proof checker for clausal unsatisfiability proofs. Our tool allows developers and users to verify solver output. Additionally, our tool can emit a reduced proof with optimal clause deletion information. A reduced clausal proof might then be be validated by a mechanically-verified proof checker.

Minimal unsatisfiable core extraction: Computing a minimal unsatisfiable subset (MUS) consists of two phases. In the first phase, called *trimming*, the input formula is solved and an antecedent graph is constructed. All original and learned clauses that are not in the cone of the empty clause are removed. This phase can substantially reduce the number of original and learned clauses, and this phase can be repeated. The second phase [11] repeats the following until a fixed point is reached. Select an original clause that is not marked as a clause in the minimal unsatisfiable core. Next, a new Boolean formula is constructed that consists of the remaining original clauses (without the selected clause) and all learned clauses that do not have the selected clause in their dependency cone. If this Boolean formula is satisfiable, the selected clause is marked as part of the minimal unsatisfiable core; otherwise, the selected clause and all learned clauses that have this clause in their dependency cone are removed.

Computing Craig interpolants: Another application from the field of model checking relies on the availability of an antecedent graph to compute *Craig interpolants* [12]. Given two satisfiable Boolean formulas A and B such that $A \wedge B$ is unsatisfiable, the interpolant I of A and B is a Boolean formula that is logically implied by A, unsatisfiable when conjoined with B, and contains only variables that are in A and B. Algorithms that compute Craig interpolants, which include recent improvements by Vizel [13], use the antecedent graph of the formula $A \wedge B$.

Optimizations

Reconstruction may result in a different antecedent graph than the one achieved during search. For example, assume that the antecedent graph in Fig 1 was produced by a SAT solver. During reconstruction, we might be able to produce a smaller antecedent graph that has fewer core clauses, core lemmas, and core arcs. An example of such an optimized antecedent graph is shown in Fig. 3.

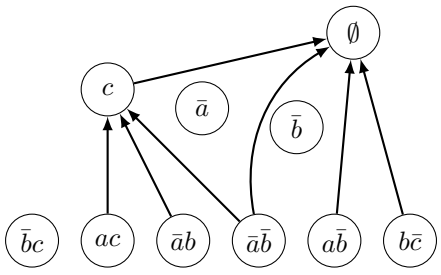


Fig. 3. Optimized antecedent graph for the example formula.

Aside from reconstruction of the antecedent graph, we also propose strategies to minimize the number of core clauses, core lemmas, or core arcs. We noticed that there is a trade-off between optimizing these different aspects (see Section VII-B).

Minimizing Core Clauses: The smaller the number of core clauses, the closer the one gets to a MUS. Hence, trying to minimize the set of core clauses during reconstruction could reduce the cost to extract a MUS.

Minimizing Core Lemmas: By reducing the number of lemmas in a clausal proof, checking costs are reduced, potentially enabling the use a mechanically-verified proof checker. Furthermore, a smaller number of lemmas can also lower the cost of MUS extraction by reducing the number of internal nodes in the antecedent graph.

Minimizing Core Arcs: The number of core arcs is related to the size of a resolution refutation of the core clauses. By minimizing the core arcs during the reconstruction, a smaller resolution refutation is obtained which can improve the speed of validating a resolution proof.

IV. Backward Reverse Unit Propagation

This section describes a method [1] to validate clausal proofs. The backward checking variant (see Section IV-B) can also be used to reconstruct an antecedent graph for a given proof. As far as we know, there is no implementation of this method available. We noticed that this method can be very expensive when used on clausal proofs from state-of-the-art CDCL solvers. Sections V-A and VI discuss two optimizations that make this method much more efficient.

A. Forward Checking

Forward checking validates *each* lemma of a proof, in the order that they were learned, by checking if they have the RUP property. Forward checking is simple to implement [1], [3], [14], relatively easy to parallelize, and can start as soon as a lemma is learned. However, this approach may check lemmas that are not required to validate a proof.

B. Backward Checking

Backward checking validates lemmas in the reverse order that they were learned. The advantage of checking a proof backward is that while validating a lemma, one can mark all the clauses that are used to determine that the lemma has the RUP property. When an unmarked lemma is encountered during a backward loop, the lemma is skipped. This can significantly reduce the checking costs by skipping lemmas during proof checking. Another advantage of backward checking is that it produces an unsatisfiable core from the original input clauses. This procedure can be used to *trim* formulas when computing minimal unsatisfiable cores.

Backward checking is more complex, however, because the checker needs to compute which clauses and lemmas were used for each lemma in a proof. Because of this complexity, it is harder to trust or verify a backward checking algorithm. Furthermore, the computation of the core clauses makes the procedure more costly. If only a few lemmas can be skipped, backward checking can be more costly than forward checking. Checking can only start when the solver has terminated, preventing an implementation that solves and checks in parallel. Finally, efficient backward checking is difficult to parallelize because lemma dependency is unknown.

Fig. 4 shows the pseudo-code of the backward checking algorithm. Its input is the original formula F and a stack of learned lemmas S. The top of the stack is the last learned clause. For a refutation, this is typically the empty clause $\emptyset$. Initially, all clauses and lemmas are unmarked (line 1). A stack without the empty clause is not a refutation (line 2). The empty clause is marked (line 3). We pop lemmas from the stack until we find a marked lemma (lines 4-6), the first of which will be the empty clause. For marked lemmas, we validate that this lemma has the RUP property with respect to the original formula and all remaining lemmas in the stack (line 7). If the check succeeds, then the clauses and lemmas that were used during BCP are marked. The algorithm succeeds if it was able to validate all lemmas in S (line 9).

```
   backwardRUP (CNF formula F, stack S of lemmas)
1    forall C ∈ F ∪ S do core [C] = 0
2    if ∅ ∉ S return "invalid refutation"
3    core [∅] = 1
4    while S is not empty do
5      L := S.pop()
6      if core [L] then
7        if BCP (F ∪ S, L̄) = "failed" then
8          return "failed"
9    return "refutation validated"
```

Fig. 4. Pseudo-code of the backward reverse unit propagation procedure.

Each lemma is validated by checking that a lemma L has the property RUP; this is performed by the BCP procedure (Fig. 5). BCP has two inputs: the set of clauses FS consisting of the original formula F and the remaining lemmas in the stack S, and the assignment τ that falsifies a lemma L (denoted by $\bar{L}$). During the procedure, a set of clauses U is maintained of all clauses that have become unit (line 1). The procedure terminates when the current assignment τ falsifies a clause in FS, by calling the *MarkCore* procedure (lines 3-4). Otherwise, it extends τ and updates U for each discovered unit clause (lines 5-7). If there are no more unit clauses and no clause has been falsified, the algorithm returns "failed" (line 8).

The *MarkCore* procedure works as follows. The falsified clause R is marked (line 1). Then, any unit clauses that were found during BCP (in stack U) are examined in reverse order. If resolution is possible between the clause C on the top of the stack and resolvent R (which was the originally falsified clause), then C is marked and the resolvent R is updated by applying the resolution step $R := R \bowtie C$. When the stack is empty, the resulting R is falsified by the input assignment τ of the BCP procedure that called *MarkCore*.

```
BCP (set of clauses FS, assignment τ)
1  U := ∅
2  forever do
3    if ∃ C ∈ FS s.t. τ(C) = f then
4      return MarkCore (U, C)
5    if ∃ C ∈ FS s.t. unit(C, τ) then
6      U.push(C)
7      τ := τ ∪ unit(C, τ)
8    else return "failed"
```

Fig. 5. Pseudo-code of the unit propagation (BCP) procedure.

```
MarkCore (unit stack U, clause R)
1  core [R] := 1
2  while U is not empty do
3    C := U.pop()
4    if C and R have exactly one clashing literal then
5      core [C] := 1
6      R := R ⋈ C
7  return "succeeded"
```

Fig. 6. The MarkCore procedure marks all clauses and lemmas that were involved in validating a lemma.

The *MarkCore* procedure is similar to the *analyzeFinal* [15] procedure that is used in CDCL solvers that support assumptions (decisions at level 0) also known as the last unique implication point.

Notice that the core arcs are not stored in this approach. This reduces the memory consumption significantly as compared to storing the antecedent graph during search. We can calculate how many core arcs would have been in the graph by summing up how often lines 1 and 5 of *MarkCore* are executed. Also, one can obtain the full antecedent graph by inserting edges from R (line 1) or C (line 5) to the current lemma L (in the backwardRUP procedure).

V. Adding Information to Clausal Proofs

The main disadvantage of clausal proof checking and trimming is the computational cost. Two methods have been proposed that add extra information in proofs to reduce the costs. The first method adds deletion information [14] (Section V-A) and the second method adds antecedents [6] (Section V-B).

A. Adding Deletion Information

The RUP checking algorithm presented in the prior section is costly for large proofs. The costs of verifying large proofs can be one-to-two orders of magnitude larger than the solving time. SAT solvers aggressively delete learned clauses during search whereas a RUP checking algorithm can only add lemmas.

In order to combat this disadvantage, we proposed to extend proof logging with clause deletion information [14]. In our proposed proof format, called DRUP (for delete reverse unit propagation), one can add lemmas to the formula (exactly in the same way as in the RUP format) and delete lemmas from the formula. Deleted lemmas have a `d` prefix. Fig. 7 shows an example CNF formula and a refutation for that formula in the DRUP format. The tool presented in this paper is the first proof checker that supports the DRUP format.

CNF formula

```
p cnf 3 6
-2  3 0
 1  3 0
-1  2 0
-1 -2 0
 1 -2 0
 2 -3 0
```

DRUP proof

```
  -2 0
d -2 3 0
  -1 0
d -1 2 0
   3 0
   0
```

Fig. 7. The CNF formula from Example 2 in the typical DIMACS format (left) and a refutation for that formula in the DRUP format (right). The literals a, $\bar{a}$, b, $\bar{b}$, c, and $\bar{c}$ are represented by 1, -1, 2, -2, 3, and -3, respectively. Whitespaces can be of any length; spacing is used to improve readability. The symbol `0` marks the end of clauses (DIMACS) and lemmas (DRUP).

```
backwardDRUP (CNF formula F, stack S of lemmas)
1  forall C ∈ F ∪ S_A do core [C] = 0
2  if ∅ ∉ S_A return "invalid refutation"
3  core [∅] = 1
4  while S is not empty do
5    ⟨L, flag⟩ := S.pop()
6    if flag ≠ "d" and core [L] then
7      if BCP ((F ∪ S_A) \ S_D, L̄) = "failed" then
8        return "failed"
9  return "refutation validated"
```

Fig. 8. Pseudo-code of the backward DRUP procedure.

We modify *backwardRUP* to account for deletion information (Fig. 8). Given a stack of of labelled lemmas S, the set S_A denotes the lemmas in S with no label, while the set S_D denotes the lemmas in S with label `d`. The top level procedure needs to be modified in three places. First, the $\emptyset$ should be in S_A (line 2). Second, we ignore other tests if the flag of a lemma is `d` (line 6). Third, all clauses in $F \cup S_A$ which are also in S_D are ignored during BCP (line 7).

B. Extending RUP with Antecedents

Another approach annotates RUP proofs with antecedent information [6]; we refer to these proofs as *extended RUP* proofs. Picosat [6] can emit extended RUP proofs. The reason why clausal proofs are expensive to validate is that the number of clauses that become unit during BCP (i.e., the set U) contains many clauses that were not required to show that the RUP property holds. In extended RUP proofs, each lemma in the proof is extended with a set of clauses that is sufficient to validate that lemma. Typically, this set contains the antecedents of the lemma. By restricting BCP to the set of clauses provided

with each lemma (instead of all clauses of $F \cup S$), one can significantly decrease the time to validate a proof.

In order to emit an extended RUP proof, a SAT solver needs to store and maintain the antecedents of all lemmas; this requires a lot of memory and can significantly reduce the solving time. Furthermore, the number of antecedents can be an order of magnitude larger than the number of literals in a lemma. Hence, extended RUP proofs can be an order of magnitude larger than RUP proofs. In contrast, DRUP proofs are usually only twice as large as RUP proofs. Finally, extended RUP proofs must contain a list of the clauses in the input formula (since they will act as antecedents). This means that a checker also needs to validate that clauses that are claimed to be in the input formula are indeed present.

VI. Preferring Core Clauses during BCP

We observed that, for many benchmarks, only a fraction of the original clauses and lemmas will be in the unsatisfiable core. To improve the speed of the checking algorithm, we considered an alternative implementation for BCP that prefers marked clauses and lemmas to unmarked clauses and lemmas. The pseudo-code of this algorithm is shown in Fig. 9.

```
CoreFirstBCP (set of clauses FS, assignment τ)
1   U := ∅
2   forever do
3     if (∃ C ∈ FS s.t. τ(C) = f) and (core [C]) then
4       return MarkCore (U, C)
5     if (∃ C ∈ FS s.t. unit(C, τ)) and (core [C]) then
6       U.push(C)
7       τ := τ ∪ unit(C, τ)
8     else if ∃ C ∈ FS s.t. τ(C) = f then
9       return MarkCore (U, C)
10    else if ∃ C ∈ FS s.t. unit(C, τ) then
11      U.push(C)
12      τ := τ ∪ unit(C, τ)
13    else return "failed"
```

Fig. 9. BCP preferring clauses and lemmas which are in the core.

Ryvchin [16] proposed postponing unit propagation on *interesting constraints*, a subset of the clauses in the original formula. We observed that the number of core clauses and core lemmas becomes smaller when one postpones unit propagation on all clauses and lemmas that are not yet in the core.

VII. Experimental Evaluation

To demonstrate the usefulness of the DRUP-trim tool[1], we experimented with it on the application benchmarks from the SAT 2009 competition. We ran our tests on a system with a 4-core Intel Core i7 2.6GHz processor, 16GB of RAM, and 1TB of disk space running MacOS X 10.8.3. Throughout this section, we use two SAT solvers: Glucose 2.2 [17] and Picosat-953 [6]. Glucose is one of the fastest SAT solvers available and won the SAT 2012 Challenge. Our approach allows for Glucose preprocessing techniques [10] in the proof format, and therefore avoids the reconstruction problems presented by Belov [18]. Picosat is the fastest solver that can emit additional results such as resolution proofs, RUP proofs, and unsatisfiable cores.

This section describes two experiments using Picosat, Glucose, and our DRUP-trim tool. In the first experiment, we evaluate the time to emit proofs and the time to extract additional results. In the second, we evaluate the effectiveness of trimming as it relates to unsatisfiable cores, reduced clausal proofs, and reduced resolution proofs.

A. Comparing solving / checking / trimming times

For our first experiment, we determined how many unsatisfiable benchmarks of the SAT 2009 application suite could be solved by Glucose and Picosat. We ran Picosat with the option to emit extended RUP proofs. We modified Glucose to emit DRUP proofs (the input format for our DRUP-trim tool). This modification is about 40 lines of code, most of which are added to support preprocessing techniques [14].

Within a timeout of 900 seconds, Glucose with DRUP logging solved 123 instances, while Picosat with extended RUP proof logging solved only 81 instances. The benchmarks solved by Picosat were a subset of the benchmarks solved by Glucose. With an even larger timeout of 9000 seconds, Picosat was only able to solve 101 out of the 123 unsatisfiable benchmarks that Glucose can solve in 900 seconds. On most of the unsolved benchmarks, Picosat exhausted memory (limit 15 Gb). We noticed that turning on the proof logging in Picosat increased the memory consumption on some benchmarks by two orders of magnitude. In contrast, proof logging in our modified version of Glucose does not require additional memory because the proof is stored directly on disk. This experiment shows the disadvantage of producing additional results within a SAT solver: one is not able to produce a proof or core due to lack of memory on several benchmarks. We observed similar problems when using the state-of-the-art MUS extraction tool Muser [19] on the same instances.

Fig. 10 shows the runtime of Glucose and Picosat with proof logging enabled and the costs to validate the proofs emitted by Glucose. In our cactus plot, the data points for each line are sorted based the y-axis. The checking costs with four different settings of our DRUP-tool are also shown. Two settings use forward checking to validate all lemmas in the proof and do not mark clauses. One setting ignores the deletion information in the proofs (denoted by RUP-checking, similar to the approach in [3]), while DRUP-trim forward uses this information (similar to the approach in [14]). The other two settings use backward checking and hence mark the clauses and lemmas in the core. The fastest setting is the one that uses the core-first BCP technique. The core-first BCP is usually faster than conventional BCP. However, for some benchmarks with hundreds of thousands of variables, conventional BCP outperforms core-first BCP. This difference is caused by the

[1] DRUP-trim is available at http://cs.utexas.edu/~marijn/drup-trim/.

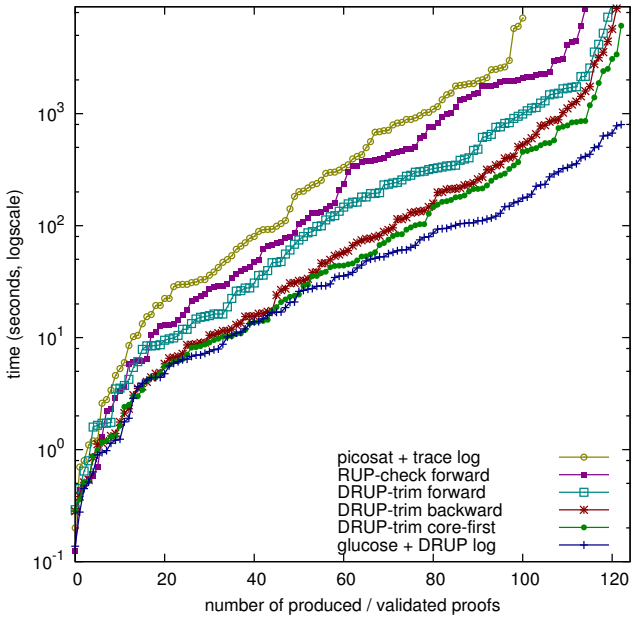


Fig. 10. Cactus plot of comparing the time to solve application benchmarks and the time to validate the emitted proofs. The plot shows the running time for the solvers Glucose with DRUP logging and Picosat with extended RUP logging. Additionally, the costs of checking the DRUP proofs with DRUP-trim are shown. Notice that the y-axis (time) uses a logarithmic scale.

structure of certain large benchmarks and is not related to size of formulas.

For the first 80 instances, the solving times recorded by Glucose and the corresponding checking times of DRUP-trim are comparable. For the next 25 instances, checking takes about twice as long as compared to solving. The checking time is only slower for a handful of instances. Yet, DRUP-trim with core-first BCP was able to check all proofs within a timeout of 900 seconds. We believe it is now feasible to check all unsatisfiability results.

B. Comparing the Quality of Trimming

We restrict the experiments to the 101 benchmarks of the SAT 2009 application suite that Picosat with extended RUP logging was able to solve within the timeout of 9000 seconds and a memory limit of 15 Gb. Recall that Glucose could solve all these instances in less than 900 seconds.

Fig. 11 shows the size of the unsatisfiable cores produced by Picosat that stores the antecedent graph and by Glucose using our DRUP-trim tool. We experimented with two variants: one with conventional BCP and one with core-first BCP. Core-first BCP not only reduces the validation cost for formulas (Fig. 10) but is also more effective in trimming the formula (Fig. 11). Furthermore, the cores produced by Picosat are larger than the ones produced by DRUP-trim using the Glucose proof. This suggests that it is not beneficial to store the antecedent graph during search in order to trim a formula.

The differences between Picosat and Glucose also have an impact on the comparison. For example, we observed that preprocessing (used in Glucose, but not in Picosat) influences the size of unsatisfiable cores produced by SAT solvers. Whether the effect is positive or negative differs from benchmark to benchmark. We compared Picosat and Glucose because they are publicly-available, state-of-the-art solvers that support resolution and clausal proofs, respectively. Picosat's poorer performance (with antecedent logging) compared to Glucose (with DRUP logging) is caused by a combination of older heuristics, lack of preprocessing, and heavier memory use.

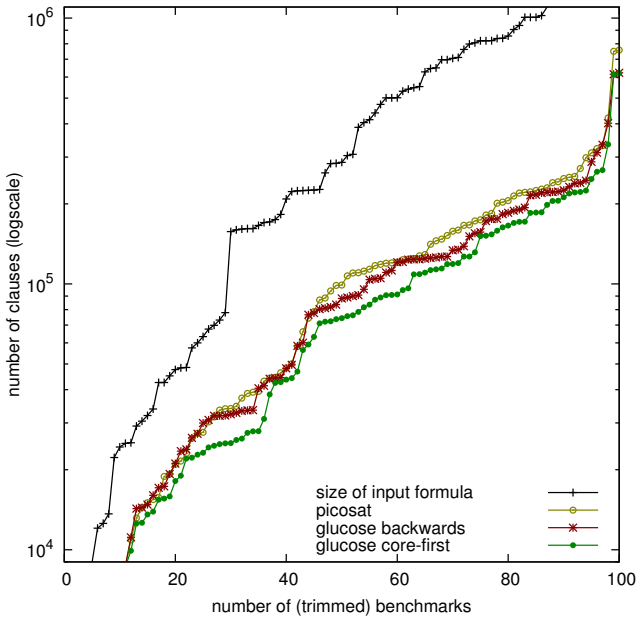


Fig. 11. Cactus plot of comparing the size (in the number of clauses) of the trimmed formulas. Notice that the y-axis (size) uses a logarithmic scale.

We modified Picosat so that it can emit DRUP proofs without storing an antecedent graph (which affects only 10 lines of code). We then compared the original version of Picosat to the modified version. The alternative DRUP approach produces smaller unsatisfiable cores (on average 11% smaller) and smaller clausal proofs (on average 21% smaller). Recall that the alternative approach does not store the core arcs and uses 1% to 10% of the memory of the original approach.

The native core approach in Picosat produces fewer core arcs compared to the approach using DRUP-trim with the core-first BCP strategy. This is not surprising: by postponing BCP on clauses and lemmas that are not in the core, several arcs to core clauses and core lemmas are produced. The number of core arcs can be substantially larger using the alternative approach (a factor four on average, mostly because of some outliers). A possible way to reduce the number of core arcs is to run the DRUP-trim tool twice. First, using core-first BCP on the original formula and proof; and second, using conventional BCP on the core clauses and the core lemmas.

Our results use the DRUP-trim tool only once. We noticed that in most cases, the core can be further reduced by applying DRUP-trim multiple times. This can be done in two ways: use the reduced clausal proof or compute a new proof for the core

clauses using a SAT solver. The best approach depends on the benchmark. For some benchmarks, a SAT solver produces a larger proof for the core clauses. In those cases, one should use the reduced proof. However, if a SAT solver is able to produce a smaller refutation for the core clauses, then those lemmas would be better for a new iteration.

VIII. Conclusion

We presented the tool DRUP-trim that can efficiently check clausal proofs and can produce additional results including unsatisfiable cores and reduced clausal proofs. DRUP-trim is the fastest clausal proof checker available. This tool also makes it more convenient to check clausal proofs and to obtain additional results. A slightly modified version of DRUP-trim was used to check the unsatisfiability results of the upcoming SAT Competition 2013.

DRUP-trim was able to verify all unsatisfiability claims of the state-of-the-art SAT solver Glucose 2.2 on the benchmarks of the SAT 2009 application suite. Other work on checking unsatisfiability results [1], [2], [3], [4], [11], [16] only shows results for selected (small) sets of benchmarks. We compared our DRUP approach with Picosat, the strongest solver that can emit resolution proofs. In contrast to emitting a DRUP proof, building a resolution proof during search can significantly increase the memory requirements of the solver. As a consequence Picosat resulted in memory outs on about 20 application benchmarks that Glucose 2.2 was able to solve in 900 seconds. This suggests that resolution proofs are not a viable method for checking the unsatisfiability results in a general setting, such as the SAT Competition.

However, our clausal-based approach does not subsume current resolution-based methods. We expect that resolution-based methods will continue to be useful for applications that require short (a few seconds) SAT solver runs. Our clausal-based approach is useful for applications that require one or more long (a few minutes) SAT runs. For a long SAT run, it is beneficial to use the latest SAT-solver technology —which is not presently available in solvers that emit resolution proofs— and avoid storing very large antecedent graphs.

One of our optimizations, the core-first BCP technique, facilitates the computation of smaller unsatisfiable cores and smaller reduced clausal proofs when compared to resolution-style methods. However, the number of core arcs increases when this technique is used. Our future work will focus on reducing the number of core arcs in DRUP-trim. This can be useful to reduce the cost of MUS extraction and interpolant tools.

The best tools for extracting minimal unsatisfiable cores, such as Muser [19], and computing interpolants, such as CNF-ITP [13], are based on resolution proofs. But there are two important drawbacks. First, similar to proof checking, some benchmarks cannot be solved when one builds a resolution proof during search. Second, none of the top-tier solvers support the emission of a resolution refutation. Current approaches [19], [11], [16], [13] rely on either Picosat or Minisat [20] which are no longer the strongest solvers. We propose to use DRUP proofs as input for tools that compute MUSes and interpolants. This makes it easy to use any SAT solver for those tools.

Acknowledgment

The authors are supported by DARPA contract number N66001-10-2-4087.

References

[1] E. I. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 2003, pp. 10 886–10 891.

[2] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *DATE*, 2003, pp. 10 880–10 885.

[3] A. Van Gelder, "Verifying RUP proofs of propositional unsatisfiability," in *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*. Springer, 2008.

[4] A. Darbari, B. Fischer, and J. Marques-Silva, "Industrial-strength certified SAT solving through verified SAT proof checking," in *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. Springer-Verlag, Sep. 2010, pp. 260–274.

[5] R. Brummayer, F. Lonsing, and A. Biere, "Automated testing and debugging of SAT and QBF solvers," in *Proceedings of SAT 2010*, ser. SAT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 44–57.

[6] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, no. 2-4, pp. 75–97, 2008.

[7] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Efficient generation of unsatisfiability proofs and cores in SAT," in *LPAR-17*, ser. LNCS, I. Cervesato, H. Veith, and A. Voronkov, Eds., vol. 5330, Springer. Springer, 2008, pp. 16–30.

[8] M. Heule, M. Järvisalo, and A. Biere, "Efficient cnf simplification based on binary implication graphs," in *SAT*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215.

[9] M. J. H. Heule and H. van Maaren, *Look-Ahead Based SAT Solvers*. Handbook of Satisfiability, IOS Press, February 2009, ch. 5, pp. 155–184.

[10] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 61–75.

[11] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2010, pp. 221–229.

[12] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification (CAV)*. Springer, 2003, pp. 1–13.

[13] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in CNF," in *Computer Aided Verification (CAV)*. Springer, 2013, p. to appear.

[14] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, "Bridging the gap between easy generation and efficient verification of unsatisfiability proofs," *Software Testing, Verification, and Reliability (STVR): Special Issue on Tests and Proofs*, 2013, accepted with minor revisions.

[15] N. Eén, A. Mishchenko, and N. Amla, "A single-instance incremental sat formulation of proof- and counterexample-based abstraction," *CoRR*, vol. abs/1008.2021, 2010.

[16] V. Ryvchin and O. Strichman, "Faster extraction of high-level minimal unsatisfiable cores," in *Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2011, pp. 174–187.

[17] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *International Joint Conference on Artifical Intelligence (IJCAI)*, C. Boutilier, Ed., 2009, pp. 399–404.

[18] A. Belov, M. Järvisalo, and J. Marques-Silva, "Formula preprocessing in MUS extraction," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013, pp. 108–123.

[19] A. Belov and J. Marques-Silva, "Accelerating MUS extraction with recursive model rotation," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 37–40.

[20] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

Simplex with Sum of Infeasibilities for SMT

Tim King* Clark Barrett* Bruno Dutertre†

*New York University †SRI International

***Abstract*—The de facto standard for state-of-the-art real and integer linear reasoning within Satisfiability Modulo Theories (SMT) solvers is the Simplex for DPLL(T) algorithm given by Dutertre and de Moura. This algorithm works by performing a sequence of local optimization operations. While the algorithm is generally efficient in practice, its local pivoting heuristics lead to slow convergence on some problems. More traditional Simplex algorithms minimize a global criterion to determine the feasibility of the input constraints. We present a novel Simplex-based decision procedure for use in SMT that minimizes the sum of infeasibilities of the constraints. Experimental results show that this new algorithm is comparable with or outperforms Simplex for DPLL(T) on a broad set of benchmarks.**

I. Introduction

The simplex algorithm introduced by Dutertre and de Moura in [1] for use in the DPLL(T) framework is the core reasoning module for linear arithmetic in nearly every state-of-the-art Satisfiability Modulo Theories (SMT) solver including CVC4, MathSAT, OpenSMT, SMTInterpol, Yices, and Z3 [2], [3], [4], [5], [6]. The algorithm—which we will call SimplexForSMT—relies on specific pivoting heuristics to search for a satisfying model or a conflict. Many pivot choices are possible and those choices can dramatically change the search for a solution. The heuristic pivot selection scheme that many SMT solvers use is based on local criteria and is potentially subject to cycling: it may return to the same basis state infinitely often. Solvers employ tactics to detect cycling, and slowly edge towards pivot-selection rules that guarantee termination, such as Bland's Rule [7], [8], [9]. Unfortunately, Bland's rule converges very slowly and is not effective on hard problems that require many pivots.

Before SimplexForSMT, earlier simplex-based approaches for SMT used repeated optimization (via an algorithm like Primal in Section III) as constraints arrived [15], [16], [17]. Since its initial publication, little work has been published on directly improving the simplex solver itself. Griggio's thesis [13] gives a number of details on implementation and additional pivoting heuristics. Most recent work on QF_LRA has focused on combining floating point and exact precision solvers [18], [19], [20].

In the more traditional setting, Simplex is used to minimize (or maximize) a linear function f. Throughout execution of the Simplex algorithm, the value of f never increases. As long as f strictly decreases, no cycling is possible. Thus, specialized techniques to prevent cycling are only required to break out of sequences of degenerate pivots, that is, pivots that do not change f. Procedures can then be designed around two different modes: a heuristic mode that is efficient in practice, and a mode for escaping degeneracy.

This paper proposes an adaptation for SMT of the sum-of-infeasibilities method from the Simplex literature [7], [8]. We call this method SOISimplex. Minimizing the sum-of-infeasibilities provides a witness function similar to f which accomplishes several things at once: it helps guide the search towards both models and conflicts; it prevents cycling; and it can be used to determine when to safely re-enable aggressive heuristics without losing termination.

In other aspects, SOISimplex is similar to the SimplexForSMT algorithm, providing similar features and having similar performance on many problems. However, its performance is noticeably better on certain problem instances that require many pivots.

The rest of the paper is organized as follows. Section II covers basic background material on SMT, DPLL(T), and linear real arithmetic. Section III describes a naive traditional primal simplex optimization routine. Section IV gives a description of SimplexForSMT. Section V then describes the new SOISimplex algorithm. Empirical results are given in Section VI, and Section VII concludes.

II. Background

The basic SMT problem is to determine whether a formula is satisfiable with respect to some fixed first-order theory T. Modern SMT solvers rely on an architecture called DPLL(T) which integrates a fast SAT solver with one or more theory solvers for specific first-order theories [10]. The SAT solver reasons about the Boolean skeleton of the formula, allowing the theory solvers to reason only about conjunctions of literals in their theory. This paper's main concern is a novel theory solver for quantifier-free linear real arithmetic (QF_LRA).

A formula in QF_LRA is a Boolean combination of atoms of the form $\sum c_j \cdot x_j \bowtie d$, where c_j, d are rational, $\bowtie \in \{=, \leq, \geq\}$, and $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is a set of variables. By using simple transformations [1], the set of constraints presented to a QF_LRA theory solver can always be written as: $T\mathcal{V} = 0 \wedge l \leq \mathcal{V} \leq u$, where T is a matrix, and l and u are vectors of lower and upper bounds on the variables. We will refer to the entry in row i and column j of T as $t_{i,j}$, and use r_i to denote the i-th row of T. We further use $l(x)$ and $u(x)$ to denote the lower and upper bound on a specific variable x. If x has no lower (upper) bound, then $l(x) = -\infty$ ($u(x) = +\infty$). The theory solver searches for an assignment $a : \mathcal{V} \mapsto \mathbb{R}$ that satisfies the constraints. If no such assignment

ISBN 978-0-9835678-3-7/13.

1: **procedure** UPDATE(j, δ)
2: $a(x_j) \leftarrow a(x_j) + \delta$
3: **for all** $i | t_{i,j} \neq 0$ **do**
4: $a(x_i) \leftarrow a(x_i) + t_{i,j} \cdot \delta$

(a) Changing $a(x_j)$ by δ ($x_j \in \mathcal{N}$)

1: **procedure** PIVOT(i, j)
2: $r_j \leftarrow r_j - \frac{1}{t_{i,j}} \cdot r_i$
3: **for all** $k | t_{k,j} \neq 0 \wedge k \neq j$ **do**
4: $r_k \leftarrow r_k + t_{k,j} \cdot r_j$
5: $\mathcal{B} \leftarrow (\mathcal{B} - \{b_i\}) \cup \{x_j\}$
6: $\mathcal{N} \leftarrow \mathcal{V} \setminus \mathcal{B}$

(b) Pivot b_i and x_j ($t_{i,j} \neq 0$)

1: **procedure** UPDATEANDPIVOT(j, δ, i)
2: UPDATE(j, δ)
3: **if** $i \neq j$ **then**
4: PIVOT(i, j)

(c) Composing Update and Pivot

Fig. 1: Algorithms for maintaining $Ta = 0$

exists, the theory solver must detect a *conflict* and provide an *explanation* (cf. [10], [1]), an infeasible subset (preferably small) of the set of constraints presented to the theory solver.

In all of the algorithms in this paper, we assume T is an $n \times n$ matrix in *tableau form*: the variables $\mathcal{V}$ are partitioned into the *basic* variables $\mathcal{B}$ and *nonbasic* variables $\mathcal{N}$ (to emphasize when a variable x_i is basic, we will write b_i as a synonym for x_i when $x_i \in \mathcal{B}$), and the i-th row of T is all zeroes iff $x_i \in \mathcal{N}$. Furthermore, for each column i such that $x_i \in \mathcal{B}$, we have $t_{k,i} = 0$ for all $k \neq i$ and $t_{i,i} = -1$. Thus, each nonzero row r_i of T represents a constraint $b_i = \sum_{x_j \in \mathcal{N}} t_{i,j} \cdot x_j$. It is sometimes convenient to use the matrix obtained by adding the identity matrix to T. We define $\tau = T + I$ and refer to the entry in row i and column j of τ as $\tau_{i,j}$. Note that on the diagonal, $\tau_{i,i} = 0$ for $b_i \in \mathcal{B}$ and $\tau_{j,j} = 1$ for $x_j \in \mathcal{N}$ (off the diagonal, $\tau_{i,j} = t_{i,j}$). The column length for a variable x_j, denoted by $|\operatorname{Col}(j)|$, is the number of nonzero entries in column j.

The algorithms in this paper work by making a series of changes to an initial assignment a until the constraints are satisfied or determined unsatisfiable. During this process, $Ta = 0$ is an invariant. To initially satisfy this invariant, one can set $a(x_i) = 0$ for all $x_i \in \mathcal{V}$. To maintain the invariant, whenever the assignment to a nonbasic variable changes, the assignments to all dependent basic variables are also updated (Fig. 1a). The main ingredient of Simplex-based procedures is the *pivoting operation* shown in Figure 1b. Pivoting takes a basic variable b_i and a nonbasic variable x_j such that $t_{i,j} \neq 0$, and swaps them: after pivoting, x_j becomes basic and b_i becomes nonbasic. Figure 1c gives the composition of the update and pivot operations, UPDATEANDPIVOT.

1: **procedure** PRIMAL(f)
2: **while** $\operatorname{Flex}(f) \neq \emptyset$ **do**
3: UPDATEANDPIVOT(PRIMALSELECT())
4: **return** $a(f)$

(a) PRIMAL(f) with a generic selection routine

1: **procedure** PRIMALSELECT
2: $S \leftarrow \emptyset$
3: **for all** $x_j \in \operatorname{Flex}(f)$ **do**
4: $S \leftarrow S \cup \langle j, k \rangle$, where $\langle | \delta_{\mathrm{B}}(j,k) |, k \rangle$ is minimal
5: select $\langle j, k \rangle \in S$ minimizing $\langle - | \operatorname{sgn}(\delta_{\mathrm{B}}(j,k)) t_{0,j} |, j \rangle$
6: **return** $\langle j, \delta_{\mathrm{B}}(j,k), k \rangle$

(b) PRIMALSELECT with a terminating variant of Dantzig's rule

Fig. 2: Primal Simplex

III. NAIVE PRIMAL SIMPLEX

The classic problem in linear optimization is to find an assignment a that satisfies the linear equalities $Ta = 0$ and the bounds $l \leq a \leq u$, and that minimizes a linear function $f = \sum_{x_k \in \mathcal{V}} c_k \cdot x_k$. The problem can be solved with the PRIMAL Simplex algorithm shown in Figure 2. It is typical to assume that the algorithm is given an initial feasible assignment as input, so that both $Ta = 0$ and $l \leq a \leq u$ are initially satisfied.

The optimization function f is treated as a special additional variable $f = x_0 = \sum_{x_k \in \mathcal{V}} c_k \cdot x_k$. We add a row and column to T (for convenience, at the top and left, indexed by 0), with $t_{0,j} = c_j$, for $1 \leq j \leq n$, $t_{i,0} = 0, 1 \leq i \leq n$, and $t_{0,0} = -1$. The entries in the new row corresponding to basic columns can be set to zero using matrix row additions (as is done in PIVOT). We can then treat f (which we use as another name for x_0 below) as a basic variable with no bounds. (Note that to instead maximize f with the same machinery, we simply minimize its negation $-f$.)

Every round of PRIMAL begins by checking whether or not f is currently at its minimum. This is done by looking at the assignments to each nonbasic variable on f's row. The value of x_j that minimizes f–call this v_j–is $u(x_j)$ if $t_{0,j}$ is negative and $l(x_j)$ if $t_{0,j}$ is positive (ignoring other constraints). If $a(x_j) = v_j$ for each nonbasic variable x_j on f's row (where $t_{0,j} \neq 0$), then the current value of f, $a(f)$, must be the minimum because we can prove $f \geq a(f)$ as follows:

$$\begin{aligned} f &= \sum_{\tau_{0,j}>0} t_{0,j} x_j + \sum_{\tau_{0,k}<0} t_{0,k} x_k \\ &\geq \sum_{\tau_{0,j}>0} t_{0,j} l(x_j) + \sum_{\tau_{0,k}} t_{0,k} u(x_k) \\ &= \sum_{\tau_{0,j}>0} t_{0,j} a(x_j) + \sum_{\tau_{0,k}<0} t_{0,k} a(x_k) = a(f) \end{aligned} \tag{1}$$

The search can then terminate. Otherwise, there is some x_j on f's row s.t. $a(x_j) \neq v_j$, and it is unclear whether $a(f)$ is at a minimum. By trying to change $a(x_j)$ for these x_j, we can at the same time hunt for an assignment that decreases $a(f)$

and search for a proof of optimality. We will call the non-basic variables on f's row whose assignments are not at their relevant bounds the *flexible* variables for this row. The set of flexible variables for an arbitrary basic variable b_i is denoted $\mathrm{Flex}(d, b_i)$ where d is a *directional* rational that is used as an implicit multiplier:

$$\begin{aligned}\mathrm{Flex}(d, b_i) = &\{x_j | d \cdot \tau_{i,j} > 0 \wedge a(x_j) > l(x_j)\} \cup \\ &\{x_k | d \cdot \tau_{i,k} < 0 \wedge a(x_k) < u(x_k)\}\end{aligned} \tag{2}$$

The parameter d allows us to choose whether to minimize or maximize b_i and will be discussed further in Sections IV and V. When $d = 1$ (as it always is in this Section), we will drop the first argument to Flex as a notational convenience. Thus, f is at its minimum when $\mathrm{Flex}(f) = \emptyset$.

To decrease the value of $a(f)$, we choose some $x_j \in \mathrm{Flex}(f)$ and determine an appropriate δ for $\textsc{Update}(x_j, \delta)$ (we discuss the strategy for picking x_j below). The direction in which we attempt to move $a(x_j)$ is determined by $t_{0,j}$: if $t_{0,j} < 0$, then we want $\delta \geq 0$ and if $t_{0,j} > 0$, then we want $\delta \leq 0$. Since the UPDATE operation must maintain the invariant $l \leq a \leq u$, the value of δ is constrained by the bounds on x_j: $l(x_j) \leq a(x_j) + \delta \leq u(x_j)$. Also, for every b_i that depends on x_j, the value $a(b_i)$ must stay within bounds: $l(b_i) \leq a(b_i) + t_{i,j} \cdot \delta \leq u(b_i)$. These cases can be unified using τ: for all k, $l(x_k) \leq a(x_k) + \tau_{k,j} \cdot \delta \leq u(x_k)$.

PRIMAL always considers $\textsc{Update}(j, \delta)$ operations that are maximal: the value of δ is selected so that at least one variable's assignment is pushed against its bound (any larger change would violate the bound). For each k, the candidate value for δ is the one that sets x_k equal to one of its bounds (which bound is determined by the sign of δ and the sign of $\tau_{k,j}$). We call these candidate values for δ the *break points* of x_j. Formally, let $\delta_{\mathrm{U}}(j, k, \alpha)$ be the amount x_j must change in order to make x_k equal to α after an UPDATE:

$$\delta_{\mathrm{U}}(j, k, \alpha) = \frac{1}{\tau_{k,j}} \left(\alpha - a(x_k)\right), \text{and}$$

$$\delta_{\mathrm{B}}(j, k) = \begin{cases} \delta_{\mathrm{U}}(j, k, l(x_k)) & t_{0,j} \cdot \tau_{k,j} > 0 \\ \delta_{\mathrm{U}}(j, k, u(x_k)) & t_{0,j} \cdot \tau_{k,j} < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The break points for x_j are all defined values of $\delta_{\mathrm{B}}(j, k)$.

In PRIMAL, for each j, we simply select k to minimize $|\,\delta_{\mathrm{B}}(j, k)|$ (ties can be broken by picking the minimum k). The operation $\textsc{Update}(j, \delta_{\mathrm{B}}(j, k))$ then maintains the invariant that no variable violates its bound. Additionally, the assignment to x_k is guaranteed to be pressed up against its bound. When $j \neq k$, x_k is a basic variable, so we can allow for (potential) future progress by pivoting x_k out of the basis and replacing it with x_j. The operation $\textsc{UpdateAndPivot}(j, \delta_{\mathrm{B}}(j, k), k)$ then maintains both the invariant $Ta = 0$ and $l \leq a \leq u$.[1] Because x_k leaves the basis, our strategy of minimizing $|\,\delta_{\mathrm{B}}(j, k)|$ to select k is called a *leaving* rule. By always selecting updates like this, PRIMAL ensures that $a(f)$ monotonically decreases.

We have just described a rule for selecting x_k given x_j, but we need an *entering* rule for selecting x_j. A simple way to ensure termination is to select the entering variable x_j with the smallest index j. This style of selecting entering and leaving variables is called Bland's rule in the literature, and its termination is a classic result of linear programming [9], [8], [7]. A better heuristic is to select x_j so as to maximize the value of $|t_{0,j}|$. This is called Dantzig's rule.[2]

The algorithm $\textsc{Primal}(f)$ in Figure 2 is a minimization routine that repeatedly selects an update and pivot until $\mathrm{Flex}(f)$ is empty and then returns the minimum value found for f.[3] The selection procedure uses a terminating variant of Dantzig's rule (it follows Dantzig's rule as long as $\delta_{\mathrm{B}}(j, k)$ is nonzero, otherwise it follow's Bland's rule). Note that when $\delta_{\mathrm{B}}(j, k) \neq 0$, the value of f strictly decreases, which makes it impossible to return to any previous state (as all previous states had larger values of f). Thus, the presense of a minimization function makes it easier to rule out cycles (the source of nonterminating runs). Termination only needs to be addressed for cases when f gets stuck and stops decreasing.

IV. SIMPLEX FOR DPLL(T)

The SIMPLEXFORSMT algorithm from [1] is tightly tuned to the DPLL(T) framework. It is designed to support incremental processing of arithmetic literals and efficient backtracking, and it computes minimal explanations in case of conflicts. Strict inequalities are encoded using an implicit infinitesimal variable δ (see [1] for details on δ-rationals).

In the DPLL(T) framework, a SAT solver incrementally sends theory literals to the theory solver. Periodically, it queries the solver about the current set of literals, expecting that the solver will either report satisfiable (with a satisfying assignment) or unsatisfiable (with a conflict). With appropriate preprocessing, we can assume that the linear equalities $Ta = 0$ are fixed (modulo pivoting) from the beginning, that all variables are initially unbounded, and that the theory literals sent by the SAT solver are of the form $x_i \leq c$ or $x_i \geq c$. The literals sent thus determine the bound constraints: $l \leq \mathcal{V} \leq u$. As in PRIMAL, the invariant $Ta = 0$ is always maintained. This is done by starting with $a(x) = 0$ for all x and by using only UPDATE to change variable assignments. The main job of SIMPLEXFORSMT then is to modify the current assignment using UPDATE until it satisfies the bounds or report a conflict if this is impossible. This is done by the SIMPLEXFORSMTCHECK routine shown in Figure 3.

This routine focuses on searching for an assignment a that satisfies $l \leq a \leq u$. We say that x is an *error variable* if a violates one of the bounds on x, and we denote by E the set

[1] When $k = j$, $\textsc{UpdateAndPivot}(j, \delta_{\mathrm{B}}(j, j), j)$ corresponds to an update without a pivot.

[2] Dantzig's rule tends to be dominated in practice by more sophisticated rules such as steepest gradient descent [9], [8], [7].

[3] For the purposes of this paper, we have ignored unbounded problems, i.e. problems where $a(f)$ can take on arbitrarily low values [9], [8], [7]. To handle this case, change the while loop condition additionally to stop once $a(f)$ is set to $-\infty$.

of error variables. Let $\mathrm{Vio}(x)$ denote the amount by which x violates its bound:

$$\mathrm{Vio}(x) = \begin{cases} l(x) - a(x) & a(x) < l(x) \\ a(x) - u(x) & a(x) > u(x) \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Thus, $\mathrm{Vio}(x)$ is nonnegative and piecewise linear, and x satisfies its bounds iff $\mathrm{Vio}(x) = 0$. Finding a satisfying assignment requires reducing each $\mathrm{Vio}(x_i)$ to 0. Locally, minimizing a $\mathrm{Vio}(x_i)$ is equivalent to minimizing $d_i \cdot x_i$ where d_i is 1 if $a(x) > u(x)$, -1 if $a(x) < l(x)$, and 0 otherwise.

In Fig. 3, the first loop ensures that the nonbasic variables satisfy their bounds (lines 3-4). The main work of the routine is the second loop which focuses on finding updates to basic variables that are in E. When $\mathrm{E} = \emptyset$, the current assignment is feasible and the search stops. Otherwise, there is some $b_i \in \mathrm{E}$.

The set $\mathrm{Flex}(d_i, b_i)$ contains the nonbasic flexible variables of row i that enable the function $d_i \cdot b_i$ to decrease. If $\mathrm{Flex}(d_i, b_i)$ is nonempty, then a variable $x_j \in \mathrm{Flex}(d_i, b_i)$ is chosen; b_i is pivoted with x_j; and the assignment to x_j is updated enough to move b_i to its violated bound. Let $\mathrm{VB}(b_i)$ denote the violated bound on b_i (either $l(b_i)$ or $u(b_i)$). Then for $x_j \in \mathrm{Flex}(d_i, b_i)$, the operation $\textsc{Update}(j, \delta_{\mathrm{U}}(j, i, \mathrm{VB}(b_i)))$ will set $a(b_i)$ to the violated bound.

If $\mathrm{Flex}(d_i, b_i)$ is empty, then the bounds on the nonbasic variables on b_i's row imply that $d_i \cdot b_i$ is at a minimum value so there is no way to satisfy $d_i \cdot b_i \leq d_i \cdot \mathrm{VB}(b_i)$ without violating some other bound. Thus, the current set of bounds is unsatisfiable. We can compute an explanation by collecting all of the contributing bounds on row i:

$$\bigwedge_{d_i \cdot \tau_{i,j} > 0} x_j \geq l(x_j) \wedge \bigwedge_{d_i \cdot \tau_{i,k} < 0} x_k \leq u(x_k) \wedge d_i \cdot b_i \leq d_i \cdot \mathrm{VB}(b_i)$$

We denote the conflict explanation generated in this fashion as $\mathrm{RC}(i)$. This explanation is minimal (if any constraint is removed, the remaining constraints are satisfiable) [11]. Upon detection, the row conflicts are added to the set of conflicts $\mathcal{C}$.

To ensure termination, it is sufficient to always select the minimum $b_i \in \mathrm{E}$ to leave the basis, and the minimum $x_j \in \mathrm{Flex}(d_i, b_i)$ (a variation of Bland's rule). However, the dominant heuristic in state-of-the-art implementations of SIMPLEXFORSMT is to instead select the $x_j \in \mathrm{Flex}(d_i, b_i)$ with minimum column length $|\mathrm{Col}(j)|$. This heuristic works quite well in practice but is not guaranteed to terminate. (The function $\mathrm{Vio}(b_i)$ will decrease to 0 for b_i but it may increase for other basic variables or for x_j.) A simple means of ensuring termination is to count the number of pivots and switch to Bland's rule once this passes a finite cap, This strategy is shown in Figure 3. The variable pc is the pivot count and, once pc reaches some threshold H, the pivot selection heuristic switches to Bland's rule. This strategy or slight variations of it are currently used by default in CVC4, MathSat [13], OpenSMT [14], Yices, Yices 2, and Z3 [12].

An improvement to the algorithm (and a contribution of this paper) can be obtained by implementing a more aggressive conflict detection. Instead of only checking the row of the first basic variable in error for a conflict, all rows are checked for conflicts. This variation (a replacement for the CHECKFORCONFLICT(I) procedure in Fig. 3 is shown in Figure 4. To implement this efficiently, we keep track of the size of $\mathrm{Flex}(\pm 1, b_i)$ for all $b_i \in \mathcal{B}$. These counts depend on the coefficients $t_{i,j}$, and the relationships $a(x_j) < u(x_j)$ and $a(x_j) > l(x_j)$. The bookkeeping for keeping these counts accurate is amortized into the theory solver operations. Conflict detection then amounts to checking when $|\mathrm{Flex}(d_i, b_i)|$ is 0 for $b_i \in \mathrm{E}$. (Only b_i that were affected in the previous iteration need to be tested for conflicts.) An evaluation of SIMPLEXFORSMT with and without this optimization (using CVC4) showed a 46% speedup on the `QF_LRA` SMT-LIB benchmarks. This optimization is on by default in CVC4.

```
1: procedure SIMPLEXFORSMTCHECK
2:   pc ← 0
3:   while ∃ x_j ∈ 𝒩 ∩ E do
4:     UPDATE(x_j, −d_j · Vio(x_j))
5:   while E ≠ ∅ ∧ 𝒞 = ∅ do
6:     UPDATEANDPIVOT(SIMPLEXFORSMTSELECT())
7:     pc ← pc +1
8:   return 𝒞 = ∅ ? Sat(a) : Unsat(𝒞)
9: procedure SIMPLEXFORSMTSELECT
10:  select b_i from E to minimize i
11:  CHECKFORCONFLICT(i)
12:  if 𝒞 ≠ ∅ then
13:    return ⟨i, 0, i⟩
14:  h ← pc < H ? 1 : 0
15:  select x_j from Flex(d_i, b_i) to minimize ⟨h·|Col(j)|, j⟩
16:  return ⟨j, δ_U(j, i, VB(i)), i⟩
17: procedure CHECKFORCONFLICT(i)
18:  if Flex(d_i, b_i) = ∅ then
19:    𝒞 ← {RC(i)}
```

Fig. 3: Check procedure for SIMPLEXFORSMT; uses a terminating selection rule and a procedure for detecting conflicts on row i

```
1: procedure CHECKALLCONFLICTS
2:   for all i|1 ≤ i ≤ n do
3:     if conflict on row i then
4:       𝒞 ← 𝒞 ∪ {RC(i)}
```

Fig. 4: Procedure that checks for all conflicts

V. SUM OF INFEASIBILITIES SIMPLEX

In this section, we introduce a Simplex-based theory solver for `QF_LRA` which we call SOISIMPLEX. Like SIMPLEXFORSMT in the previous section, it is designed to search for both conflicts and satisfying assignments in the context of a DPLL(T) search. It attempts to address the troubling lack of a straightforward global criterion for progress in SIMPLEXFORSMT by introducing a function to minimize. The function minimized is the sum of infeasibilities of all of the variables.

```
1: procedure SOICheck
2:   while ∃ x_j ∈ 𝒩 ∩ E do
3:     Update(x_j, −d_j · Vio(x_j))
4:   while Flex(f) ≠ ∅ ∧ 𝒞 = ∅ do
5:     UpdateAndPivot(SOISelect())
6:   if 𝒞 ≠ ∅ then
7:     return Unsat(𝒞)
8:   else if E = ∅ then
9:     return Sat(a)
10:  else
11:    return Unsat(SoiQE) (Sec. V-B)
12: procedure SOISelect
13:   CheckAllConflicts()
14:   if 𝒞 ≠ ∅ then
15:     return ⟨1, 0, 1⟩
16:   S ← ∅
17:   for x_j ∈ Flex(f) do
18:     L ← ∅
19:     for all k|k = j ∨ t_{k,j} ≠ 0 do
20:       L ← L ∪ {⟨δ_U(j, k, l(x_k)), k⟩}
21:       L ← L ∪ {⟨δ_U(j, k, u(x_k)), k⟩}
22:     select ⟨δ, k⟩ ∈ L to minimize ⟨ΔVio(j, δ), |δ|, k⟩
23:     S ← S ∪ ⟨j, δ, k⟩
24:   select ⟨j,δ,k⟩ ∈ S minimizing ⟨sgn(ΔVio(j, δ)) · |t_{0,j}|, j⟩
25:   return ⟨j, δ, k⟩
```

Fig. 5: SOICheck and selection rules for SOISimplex

For a given assignment, the sum of infeasibilities is given by: $\mathrm{Vio}(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}(x)$. Let Vio_F be the result of replacing $a(x)$ by x in the definition of Vio. The optimization function can be written as: $\mathrm{Vio}_F(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}_F(x)$. Minimizing the sum of infeasibilities is a standard technique for finding an initially feasible assignment for linear programs [7], [8].

We assume the same setup as in the previous section: we start with a fixed (modulo pivoting) tableau and a satisfying assignment a, and then the SAT solver sends a set of literals that determine the upper and lower bounds for the variables. The theory solver must provide a check routine that either reports satisfiable (with a satisfying assignment) or unsatisfiable (with a conflict). The main loop for SOISimplex uses essentially the same machinery to minimize $\mathrm{Vio}_F(\mathcal{V})$ as was used in Primal for minimizing a linear function f. However, there are a number of complications caused by the fact that $\mathrm{Vio}_F(\mathcal{V})$ is only piecewise linear instead of linear. The majority of this section is devoted to handling these challenges.

Because we cannot represent the optimization function $\mathrm{Vio}_F(\mathcal{V})$ directly in the tableau, we use a linearized approximation. First note that that $\mathrm{Vio}(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}(x) = \sum_{x_i \in \mathcal{V}} d_i \cdot (a(x_i) - \mathrm{VB}(x_i))$. In some neighborhood of $a(x_i)$, the value of $d_i \cdot \mathrm{VB}(x_i)$ will be constant. Discarding this term and replacing $a(x_i)$ with x_i results in the function $f(\mathcal{V}) = \sum_{x_i \in \mathcal{V}} d_i \cdot x_i$. Note that the function still depends on the current assignment (which determines d_i), but for a given assignment, the function is linear. We can substitute for the basic variables and rearrange the sums to get:

$$f = \sum_{x_j \in \mathcal{N}} \left(\sum_{x_i \in \mathcal{V}} d_i \tau_{i,j} \right) \cdot x_j.$$

We use this function in roughly the same way we used f in Primal: it is the 0th variable and it is always basic. To compute the tableau row for f, we simply compute coefficients for each nonbasic variable x_j by adding, for each row i, the entry in column j multiplied by the directional multiplier d_i. The computed coefficients depend on d_i and thus have to be updated every time the assignment changes. This can be implemented efficiently by instrumenting Update to detect when d_i changes to d_i' for some i. When this happens, we update f's row (r_0) as follows: $r_0 \leftarrow r_0 + (d_i' - d_i) \cdot \tau_i$ (where τ_i is the i-th row of τ).

The check procedure for SOISimplex is given in Fig. 5. It iterates while: no row contains a conflict ($\mathcal{C} = \emptyset$), and there is a nonbasic variable on f's row with slack ($\mathrm{Flex}(f) \neq \emptyset$). If $\mathcal{C} \neq \emptyset$, then SimplexForSMTCheck safely terminates with the discovered conflict. If $\mathrm{Flex}(f)$ and E are empty, the current assignment is satisfying. Otherwise, $\mathrm{E} \neq \emptyset$, $\mathrm{Flex}(f) = \emptyset$, and f is at a minimum. Section V-B discusses extracting a conflict explanation with the SoiQE procedure.

As in the Primal algorithm, the selection procedure iterates over all $x_j \in \mathrm{Flex}(f)$. The leaving rule considers x_j as well as every basic variable b_k where $t_{k,j}$ is nonzero. We consider two possible updates (break points) for each such variable: one which sets it to its upper bound and one which sets it to its lower bound. Unlike Primal, we consider updates for which some new basic variable could become violated. However, we still ensure that global progress is made. We denote by $\Delta\mathrm{Vio}(j, \delta)$ the amount that $\mathrm{Vio}(\mathcal{V})$ would change if we were to change the current assignment by executing $\textsc{Update}(j, \delta)$. From all of the possible leaving variables and updates, we then select the pair for which $\Delta\mathrm{Vio}(j, \delta)$ is minimal (equivalently, the pair that reduces the value of $\mathrm{Vio}(\mathcal{V})$ the most). Section V-A describes how to efficiently compute the values for $\Delta\mathrm{Vio}(j, \delta)$. We also show in that section that for each x_j, there is always a choice of $\langle \delta, k \rangle$ such that $\Delta\mathrm{Vio}(j, \delta) \leq 0$. This ensures that $\mathrm{Vio}(\mathcal{V})$ monotonically decreases. Tie breaking for the leaving rule is done by selecting the minimum value of $|\delta|$ and then the minimum variable index k. The motivation for the former is discussed in subsection V-C.

The entering rule selects between candidate triples $\langle j, \delta, k \rangle$ for $x_j \in \mathrm{Flex}(f)$. Any triple for which $\Delta\mathrm{Vio}(j, \delta)$ is negative ensures that SOISimplex is making progress. This allows for SOISimplex to treat $\mathrm{Vio}(\mathcal{V})$ in a manner analogous to $a(f)$ in Primal. Following our modified Dantzig's rule, we select the entering variable with the largest coefficient so long as it decreases $\mathrm{Vio}(\mathcal{V})$ with ties being broken by selecting the variable with the smaller index.

We show how SOISimplex works using the simple example shown in Fig. 6. With the given assignment, the bound $x_1 \geq 3$ is violated, and $\mathrm{Vio}(\mathcal{V}) = 2$. The variable x_2 is

$$
\begin{array}{rrcl}
T: & f & = -2 \cdot x_2 + x_3 & \\
 & x_1 & = 2 \cdot x_2 - x_3 & \\
3 \leq & x_1 & \leq 7 & a(x_1) = 1 \\
 & x_2 & \leq 3 & a(x_2) = 1 \\
1 \leq & x_3 & & a(x_3) = 1
\end{array}
$$

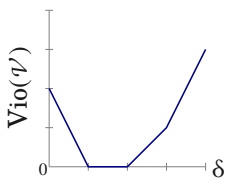


Fig. 6: Simple example showing $\mathrm{Vio}(\mathcal{V})$ after UPDATE(x_2, δ)

flexible, and we examine it for updates. The break points for x_2 are at $\delta \in \{1, 2, 3\}$, and correspond to changes to x_2 that respectively set x_1 to its lower bound, x_2 to its upper bound, and x_1 to its upper bound. Figure 6 shows how the value of $\mathrm{Vio}(\mathcal{V})$ changes if x_2 is updated by δ. For $\delta \in \{1, 2\}$, $\Delta\mathrm{Vio}(2, \delta) = -2$ and $\mathrm{Vio}(\mathcal{V})$ will become 0. Because of the tie-break on $|\delta|$, the pair $\langle \delta.k \rangle = \langle 1, 1 \rangle$ is selected, and then the triple $\langle 2, 1, 1 \rangle$ is returned. After the call to UPDATE, the algorithm terminates with a satisfying solution.

A. Computing $\Delta\mathrm{Vio}(j, \delta)$

To implement line 22 of SOISELECT, we must compute the values of $\Delta\mathrm{Vio}(j, \delta)$ for every break point δ. We use the fact that the function Vio_F is linear between break points and that the slopes of these linear segments can be computed. Let Δ be a increasing sorted list of the positive δ values in L, and let $\delta_0 = 0$: $0 = \delta_0 < \delta_1 < \ldots$. Let κ_i be the set of values of k that are paired with δ_i in L. We proceed as follows. We know that $\Delta\mathrm{Vio}(j, 0) = 0$ and that the slope β_0 as δ increases from 0 is $t_{0,j}$. Now, we can compute:

$$\Delta\mathrm{Vio}(j, \delta_i) = \Delta\mathrm{Vio}(j, \delta_{i-1}) + \beta_{i-1} \cdot (\delta_i - \delta_{i-1}).$$

Furthermore, we know that at δ_i, each variable x_k (for $k \in \kappa_i$) transitions to satisfying its bound or violating its bound, meaning that d_k will change at δ_i to some d'_k. This change can be used to compute the slope β_i for the next segment: $\beta_i = \beta_{i-1} + \sum_{k \in \kappa_i} (d'_k - d_k) \cdot \tau_{k,j}$. Continuing this walk over increasing values of δ computes $\Delta\mathrm{Vio}(j, \delta)$ for all $\delta \geq 0$. Another analogous pass can be done to compute the $\Delta\mathrm{Vio}(j, \delta)$ values for negative δ values. A number of nice properties follow from the above computation, including the the following lemma:

Lemma 1. *For each $x_j \in \mathrm{Flex}(f)$, there is some pair $\langle \delta, k \rangle \in \mathrm{L}$ such that $\Delta\mathrm{Vio}(j, \delta) \leq 0$.*

Proof. If $\delta = 0$ is a break point, then $\Delta\mathrm{Vio}(j, 0) \leq 0$. Now assume 0 is not a break point. The x_j's considered are on f's row so $t_{0,j} \neq 0$. If $t_{0,j} > 0$, there must exist some $d_i \cdot \tau_{i,j} > 0$. So there exists a negatively-valued break point, $\delta_{\mathrm{U}}(j, i, \mathrm{VB}(i))$. Let δ be the negative break point closest to 0. We know that $\Delta\mathrm{Vio}(j, \delta) = 0 + t_{0,j} \cdot \delta < 0$. Similarly, if $t_{0,j} < 0$, then $\Delta\mathrm{Vio}(j, \delta) < 0$ for the minimal positive δ. □

The proof further suggests that it is sufficient to consider either just the negative or just the positive values of δ (depending on the value of $t_{0,j}$) without affecting correctness.

B. Conflicts with Multiple Rows

If $\mathrm{Flex}(f) = \emptyset$, $\mathcal{C} = \emptyset$ (i.e. no single row produces a conflict) but $\mathrm{E} \neq \emptyset$, we can still detect a conflict and derive an explanation as follows. Similar reasoning to that in (1) can be used to show that the sum of the assignments for the variables in E is strictly greater than the sum of their violated bounds:

$$\sum_{b_i \in \mathrm{E}} d_i \cdot a(b_i) > \sum_{b_i \in \mathrm{E}} d_i \cdot \mathrm{VB}(i).$$

So the bounds on the nonbasic variables in f's row and the basic variables in error cannot together be satisfied. This allows us to extract the following conflict explanation:

$$\bigwedge_{\tau_{0,j} > 0} x_j \geq l(x_j) \wedge \bigwedge_{\tau_{0,k} < 0} x_k \leq u(x_k) \wedge \bigwedge_{b_i \in \mathrm{E}} d_i b_i \leq d_i \, \mathrm{VB}(i)$$

Explanations constructed like this may not be minimal. However, we observe that for any subset S of E, if we construct the function $f_S = \sum_{x_j \in \mathcal{N}} \left(\sum_{b_i \in S} d_i \cdot t_{i,j} \right) \cdot x_j$, and $\mathrm{Flex}(f_S) = \emptyset$, then we can extract a smaller explanation using only the rows corresponding to basic variables in S. We use a number of heuristics and a straightforward adaptation of the QuickXplain algorithm [21] to attempt to find a minimal subset S that still generates a conflict (without additional Simplex search). Most of the time a conflict can be found with $|S| = 2$. In this case, the explanation is guaranteed to be minimal.

C. Termination

The termination of SOISIMPLEX is again based on the termination of Bland's rule. Suppose that SOISIMPLEX does not terminate. There are only a finite number of possible assignments that can be considered as the number of variables is finite, and every change to the assignment assigns a variable x_j to either $u(x_j)$ or $l(x_j)$. Because the value of $\mathrm{Vio}(\mathcal{V})$ is determined by the assignment and monotonically decreases, any nonterminating execution must have an infinite tail during which $\mathrm{Vio}(\mathcal{V})$ is unchanged and the update selected, $\langle j, \delta, k \rangle$ is such that $\Delta\mathrm{Vio}(j, \delta) = 0$. As was shown in the proof of Lemma 1, if the minimal $\Delta\mathrm{Vio}(j, \delta)$ found is 0, then $\delta = 0$ must be a break point. The leaving rule enforces that the δ selected minimizes the tuple $\langle \Delta\mathrm{Vio}(j, \delta), |\delta|, k \rangle$. So in the tail of a nonterminating execution $\Delta\mathrm{Vio}(j, \delta) = 0$ and $\delta = 0$ at every step. Thus after this point, no variable is changing in assignment and no variable changes its relationship to its bounds. Every leaving and entering variable is then selected based on picking the minimum index. The argument that PRIMAL cannot cycle under Bland's rule can then be directly applied. We refer readers interested in the proof of the termination of Bland's rule to [7], [8], [9].

D. Heuristics and $\mathrm{Vio}(\mathcal{V})$

Instead of examining all $x_j \in \mathrm{Flex}(f)$ for the best candidate, we can instead just look at heuristically many candidates. The search can stop once a candidate has been found that makes progress (i.e. $\Delta\mathrm{Vio}(j, \delta) < 0$). Further, there is more freedom in selection heuristics than we have shown here. In particular, one can use any heuristic desired until no

progress has been made for a while. CVC4's implementation for example uses a heuristic that prefers shorter columns until progress stalls and then uses Bland's rule.

During the calculation of break points, it is possible to determine if pivoting x_j with b_i would result in a row conflict on x_j's new row in $O(1)$ time by using the $|\operatorname{Flex}(\pm 1, b_i)|$ values. Such selections are always prefered. CVC4's selection also heuristically prefers the set E to be as small as possible.

VI. Experimental Results

In this section we describe two experiments. In the first, we compare CVC4 against itself using two different sets of options.[4] The first set of options uses the default solver, an implementation of SimplexForSMT (which is a bit better than the version that won the `QF_UFLRA` division—which includes `QF_LRA`—of SMT-COMP 2012 [22]). The second set of options enables a new implementation of SOISimplex. The two configurations of CVC4 are run with most other heuristics disabled so that the comparison is an accurate reflection of the performance of the two algorithms as described in this paper.[5] The comparison is done on the `QF_LRA` benchmarks from the `SMT-LIB` library [23] as well as a new family of benchmarks from biological modeling, `latendresse` [24]. The `latendresse` family of benchmarks is a set of problems that originated from an analysis of biochemical reactions using the flux-balance analysis method.[6] The `miplib` and `latendresse` families are of particular interest as they contain the only timeouts in these experiments. These problems are characterized by relatively little propositional structure, and a large and relatively dense input tableau. All of the experiments were conducted on a 2.66GHz Core2 Quad running Debian 7.0 with a time limit of 1000 seconds. Every example stays below a memory limit of 2GB. Overall, SOISimplex solves 636 while SimplexForSMT solves only 629. Interestingly, SOISimplex is slightly slower on the `SMT-LIB` benchmarks (see Fig. 7), and even solves one fewer benchmark (the satisfiable `miplib` benchmark `fixnet-7000.smt2`), but solves all of the `latendresse` benchmarks while SimplexForSMT times out on 8 of them.

To understand these results better, we recorded how many pivots were done (for both algorithms) during each call to the respective check routines (for benchmarks that both algorithms are able to solve). For the `SMT-LIB` benchmarks, almost all queries sent to the theory solver are "easy" for the simplex solvers (both SimplexForSMT and SOISimplex). Table I shows, for given numbers of pivots (or ranges of numbers of pivots), the number of calls to check whose pivot count is in that range. The maximum number of pivots for any single call to check is 2238. The number of pivots

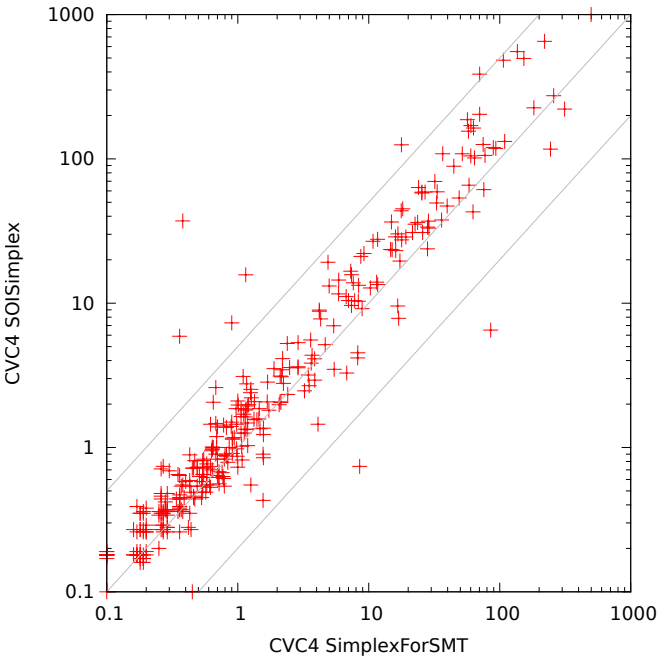


Fig. 7: Log-scaled running times (sec.) for experiment 1 on the `QF_LRA` benchmarks from `SMT-LIB`.

is generally very low and on average, SOISimplex uses fewer pivots than SimplexForSMT. The 8 timeouts by SimplexForSMT on `latendresse` have a very different signature. Each of them times out in the middle of a very long SimplexForSMTCheck call performing thousands of pivots. On average, the interrupted SimplexForSMTCheck routines had performed 18263 pivots and had been running 937s [/1000s]. This first experiment confirms our expectation that SOISimplex is effective at reducing the number of pivots required to solve a problem.

For the second experiment, we compare the same two algorithms in CVC4 against a number of state-of-the-art `QF_LRA` solvers. For this experiment, we enable a number of additional CVC4 options (those used in the SMT-EVAL 2013 run script) which are beyond the scope of this paper and which significantly improve performance (for both algorithms) on the `miplib` and `latendresse` benchmarks. These are disabled in the first experiment to better understand the relative strengths of the two algorithms on their own. The other solvers we compare with are: Z3 4.1.2 [4], mathsat 5.2.3 [3], yices 2.1.1, and OpenSMT 1.0.1[6]. Table II contains a summary of the number of problems solved by each solver and the cumulative time taken on the solved instances for the three families of benchmarks: all `SMT-LIB QF_LRA` benchmarks, the `miplib` family from `QF_LRA`, and the `latendresse` benchmarks.[7] The second experiment shows that the strongest overall solving strategy is obtained by using SOISimplex.

[4]Experiments were run using the submission to SMT-EVAL 2013: CVC4 version 1.2, available at github.com/CVC4/CVC4/tree/smteval2013.

[5]Both solvers are run with `--new-prop --no-restrict-pivots`. SOISimplex is run with the additional flag `--use-soi`. The `--no-restrict-pivots` flag disables stopping simplex after K pivots at non-leaf SimplexForSMTCheck calls ($K = 200$ by default).

[6]These benchmarks are available at cs.nyu.edu/~taking/soi.tgz and have been submitted for inclusion into `SMT-LIB`'s `QF_LRA` family.

[7]OpenSMT gave no answer on the `latendresse` benchmarks.

Range for n	0	1	$[2, 10]$	$[11, 100]$	$[101, 1000]$	$[1000, 2238]$	total
Number of calls to SIMPLEXFORSMTCHECK with n pivots	32832424	645473	896659	174743	2362	7	34551668
$\sum$ Total number of pivots performed by these calls	0	645473	3677258	3628577	479386	10173	8440867
Number of calls to SOICHECK with n pivots	30475287	924639	1008398	130167	655	0	32539146
$\sum$ Total number of pivots performed by these calls	0	924639	3900190	2366117	126506	0	7317452

TABLE I: Number of pivots per call to check for experiment 1.

	CVC4 SOISIMPLEX		CVC4 SIMPLEXFORSMT		Z3		yices2		mathsat		opensmt	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
QFLRA (634)	627	5621.29	625	5523.15	620	5582.46	619	5300	608	8043	597	17261
miplib (48)	35	641.3	33	1760	28	1158.42	27	1616	19	3049	21	1509
latendresse (18)	18	883.53	18	205	8	17.98	10	103.38	10	94.73	-	-

TABLE II: Running time and number of problems solved for experiment 2.

VII. CONCLUSION

The authors believe these experiments demonstrate both the strength and weakness of SIMPLEXFORSMT's local optimization criteria. It is good at keeping the amount of work small in the context of a DPLL(T) style search. The local optimization criteria requires little analysis and is quite an efficient heuristic for many SMT problems; however, its global convergence is questionable on large and hard examples. SOISIMPLEX adds a global optimization criterion and appears to be more robust for large and hard examples, but this comes with the cost of some additional analysis during pivot selection. Future work will explore how to heuristically take advantage of the best characteristics of both algorithms.

ACKNOWLEDGEMENTS

We'd like to thank the other members of the NYU ACSys research group for their many contributions to CVC4. This work was funded in part by NSF Grants CCF-0644299, CNS-0917375, and NASA Cooperative Agreement NNA10DE73C.

REFERENCES

[1] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *CAV 2006*, LNCS 4144. Springer-Verlag, August 2006, pp. 81–94.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV 2011*, LNCS 6806. Springer-Verlag, 2011, pp. 171–177.

[3] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS 2013*, LNCS 7795. Springer-Verlag, 2013.

[4] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS 2008*, LNCS 4963. Springer-Verlag, 2008, pp. 337–340.

[5] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: an interpolating smt solver," in *Model Checking Software (SPIN Workshop 2012)*, LNCS 7385. Springer-Verlag, 2012, pp. 248–254.

[6] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *TACAS 2011*, LNCS 6605. Springer-Verlag, 2011, pp. 150–153.

[7] P. E. Gill, W. Murray, and M. H. Wright, *Numerical linear algebra and optimization. Vol. 1*. Redwood City, CA: Addison-Wesley Publishing Company, 1991.

[8] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[9] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, 1989.

[10] R. Niewenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *JACM*, vol. 53, no. 6, pp. 937–977, November 2006.

[11] B. Dutertre and L. de Moura, "Integrating Simplex with DPLL(T)," Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01, May 2006.

[12] L. de Moura, N. Bjørner, and C. Wintersteiger, "Z3 Source Code v4.3.1 `select_pivot`," http://z3.codeplex.com/SourceControl/changeset/view/89c1785b73225a1b363c0e485f854613121b70a7#src/smt/theory_arith_core.h.

[13] A. Griggio, "An Effective SMT Engine for Formal Verification," Ph.D. dissertation, DISI - University of Trento, December 2009.

[14] R. Bruttomesso, S. Fulvio Rollini, N. Sharygina, and A. Tsitovich, "OpenSMT Source Code r64," http://opensmt.googlecode.com/svn/trunk/src/tsolvers/lrasolver/LRASolver.C.

[15] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *JACM*, vol. 52, no. 3, pp. 365–473, May 2005.

[16] H. Rueß and N. Shankar, "Solving linear arithmetic constraints," SRI International, Tech. Rep. SRI-CSL-04-01, 2004.

[17] G. Badros, A. Borning, and P. Stuckey, "The Cassowary linear arithmetic constraint solving algorithm," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 8, no. 4, pp. 267–306, December 2001.

[18] D. Monniaux, "On using floating-point computations to help an exact linear arithmetic decision procedure," in *CAV 2009*, LNCS 5643. Springer-Verlag, 2009, pp. 570–583.

[19] D. Caminha Barbosa de Oliveira and D. Monniaux, "Experiments on the feasibility of using a floating-point simplex in an SMT solver," in *Workshop on Practical Aspects of Automated Reasoning (PAAR)*. CEUR Workshop Proceedings, 2012.

[20] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Sat modulo the theory of linear arithmetic: Exact, inexact and commercial solvers," in *SAT 2008*, pp. 77–90.

[21] U. Junker, "QuickXplain: Conflict detection for arbitrary constraint propagation algorithms," in *IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

[22] R. B. Bruttomesso, D. Cok, and A. Griggio, "Smt-comp 2012," Jun. 2012. [Online]. Available: http://smtcomp.sourceforge.net/2012/

[23] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[24] M. Latendresse, M. Krummenacker, M. Trupp, and P. D. Karp, "Construction and completion of flux balance models from pathway databases," *Bioinformatics*, vol. 28, p. 38896, 2012.

Efficient MUS Extraction with Resolution

Alexander Nadel[1], Vadim Ryvchin[1,2], Ofer Strichman[2]
[1]Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
{alexander.nadel, vadim.ryvchin}@intel.com
[2]Information Systems Engineering, Technion, Israel ofers@ie.technion.ac.il

***Abstract*—We report advances in state-of-the-art algorithms for the problem of Minimal Unsatisfiable Subformula (MUS) extraction. First, we demonstrate how to apply techniques used in the past to speed up resolution-based Group MUS extraction to plain MUS extraction. Second, we show that *model rotation*, presented in the context of *assumption-based* MUS extraction, can also be used with *resolution-based* MUS extraction. Third, we introduce an improvement to rotation, called *eager rotation*. Finally, we propose a new technique for speeding-up resolution-based MUS extraction, called *path strengthening*. We integrated the above techniques into the publicly available resolution-based MUS extractor `HaifaMUC`, which, as a result, now outperforms leading MUS extractors.**

I. Introduction

Given an unsatisfiable formula in Conjunctive Normal Form (CNF), an *Unsatisfiable Subformula* (or *Unsatisfiable Core*; hereafter, *US*) is an unsatisfiable subset of its clauses. A *Minimal Unsatisfiable Subformula (MUS)* is a US such that removal of any of its clauses renders it satisfiable. The problem of finding a MUS is an active area of research [1]–[6].

The basic algorithm used in modern MUS extractors such as `MUSer2` [7] and `HaifaMUC` [3] is as follows. In the initial *approximation stage* the algorithm finds a not-necessarily-minimal US S with one or more invocations of a SAT solver [8], [9]. It then applies the following *deletion-based* iterative process over S's clauses until S becomes a MUS. Each iteration removes a *candidate* clause c from S and invokes a SAT solver. If the resulting formula is satisfiable, c must belong to the MUS, so c is returned to S and marked as *necessary*. Otherwise c is removed from S. In addition, the following two optimizations are commonly applied. First, incremental SAT solving [10], [11] is used across all SAT invocations. Second, when a clause c is found to be not necessary, one can remove from S not only c, but all the clauses (if any) omitted from the new core found by the SAT solver. This latter technique is called *clause set refinement* in [6]. The algorithm we have described up to here was introduced in [12] and improved in [2], while the idea of removing constraints one by one in order to get a minimally infeasible set can be traced back to [13], [14]. See [2] for a more detailed presentation of the algorithm and [1] for an overview of various approaches to MUS extraction.

It was demonstrated in [2] that the approach we have described can be implemented using either a resolution-based or an assumption-based algorithm. The former relies on the resolution proof maintained by the SAT solver for detecting the core at each step, while the latter adds a new *assumption literal* to each clause and detects the core using these assumptions. It was shown in [2] that the resolution-based approach to MUS extraction is faster than the assumption-based approach mainly because of the overhead of maintaining assumption literals.

Various applications require finding a MUS with respect to user-given groups of clauses [2], [15], called *interesting constraints*, while clauses that do not belong to any interesting constraint are called the *remainder*. The resulting problem is called *Group MUS (GMUS)* extraction (or *high-level MUS* extraction). It was shown in [2] that the approach we described for plain MUS extraction can be applied to GMUS extraction as well. Furthermore, it was shown in [3] that the resolution-based approach to GMUS extraction can be improved considerably by directing the search to ignore the interesting constraints and to use the remainder and the necessary clauses instead whenever possible. We call the techniques of [3] *MUS-biased search*.

The first contribution of this paper is in showing that MUS-biased search can be applied to plain MUS extraction. The key observation is that while there are no *remainder* clauses in plain MUS extraction, *necessary* clauses can still be used for MUS-biased search after the approximation stage.

A recent essential enhancement to the plain MUS extraction algorithm we have described is *model rotation* (or, simply, *rotation*) [4], [6], [16]. Rotation was proposed in the context of assumption-based MUS extraction. After implementing rotation, the resulting assumption-based MUS extractor `MUSer2` outperformed the state-of-the-art resolution-based MUS extractor `HaifaMUC`. It is sometimes postulated that rotation gives the assumption-based approach an edge over the resolution-based approach (cf. [5]).

The second contribution of this paper is thus in showing that model rotation can be integrated into the resolution-based approach. The paper's third contribution is an improvement to model rotation, called *eager rotation*, detailed in Sect. II-B.

The fourth contribution of our paper is called *path strengthening*. It is a generalization of a technique proposed in [17] and later called *redundancy removal* in [6] and implemented in `MUSer2` [7]. Redundancy removal adds the literals of $\neg c$ (where c is the candidate clause) as assumptions when checking the satisfiability of $S \setminus c$, because since S is known to be unsatisfiable, then $S \setminus c$ and $(S \setminus c) \wedge \neg c$ are equisatisfiable. Path strengthening, on the other hand, adds as assumptions the literals of $\neg c, \neg c_1, \ldots, \neg c_m$ for some $m \geq 0$, where the sequence of clauses $c, c_1, \ldots, c_m$ constitutes the longest common prefix of all paths in the resolution proof from c to

ISBN 978-0-9835678-3-7/13.

the empty clause. Further details about path strengthening are provided in Sect. II-C.

We integrated our algorithms into the resolution-based MUS extractor `HaifaMUC`. We show in Sect. III that, as a result, `HaifaMUC` now outperforms the leading MUS extractors `MUSer2` and `Minisatabb` [18]. `Minisatabb` improves `MUSer2` considerably based on the idea of replacing blocks of assumptions with new variables [18].

II. The Algorithms

A. MUS-Biased Search

We will now describe how we adapted optimizations **A-D** of the GMUS-oriented techniques proposed in [3] to plain MUS extraction (we also tried adapting optimizations **E-G** [3], but their impact on plain MUS extraction was negligible). We denote the set of necessary input clauses by M. We call an input clause c *interesting* if it belongs to $S \setminus M$ (i.e., c can still serve as a candidate). A learned clause is marked as *interesting* if it is derived using at least one interesting clause; otherwise it is marked as *necessary*. If an interesting learned clause participates in the proof, then the core includes its interesting roots; this is undesirable since we are trying to minimize the core. Most of our techniques are therefore targeted at biasing the solver towards learning *necessary* rather than *interesting* clauses. This is the reason that we call them, jointly, *MUS-biased search*. An exception is the first optimization below, which is focused on reducing the amount of memory used to store the proof.

A. *Maintain partial resolution proofs*. There is no need to store in the proof any clauses identified as necessary, since the algorithm does not need to work with these clauses explicitly anymore. Hence, we discard from the proof all the clauses that emanate exclusively from M.

B. *Perform selective clause minimization*. Clause minimization [19] is a technique for shrinking conflict clauses. Specifically, if a conflict clause c contains two literals l_1, l_2 such that $l_1 \rightarrow l_2$ because of the rest of the formula, then l_2 can be removed from c. The disadvantage of this technique in our context is that it may reclassify c from 'necessary' to 'interesting', if the implication $l_1 \rightarrow l_2$ depends on an interesting clause. This in turn may increase the size of the core later on as explained above. Hence our optimization does not apply clause minimization if it leads to such a reclassification. In other words we prefer a longer conflict clause if this enables us to maintain its classification as a necessary clause.

C. *Postpone propagation over interesting clauses*. Perform Boolean Constraint Propagation (BCP) on necessary clauses first, with the aim of learning a necessary clause when possible.

D. *Reclassify interesting clauses*. When an interesting clause c becomes necessary, look for any clauses in the resolution derivation that were derived from c that also become necessary (that is, were derived solely from necessary clauses) and reclassify them.

Note that while these optimizations improve GMUS extraction even during the approximation stage owing to the availability of remainder clauses, their impact on plain MUS extraction begins only during the minimization stage, when there are enough necessary clauses (which, like remainder clauses, must be in the proof). Indeed we demonstrate in Sect. III that optimization **B** is not cost-effective before there is a significant number of necessary clauses, which is the reason that we invoke it starting from the 2nd satisfiable iteration.

B. Eager Model Rotation

Model rotation can improve deletion-based MUS extraction by searching for additional clauses that should be marked as necessary *without* an additional SAT call. Suppose, for example, that for an unsatisfiable set S, $S \setminus c$ is satisfiable. Consequently c is marked as necessary. Let h be the satisfying assignment. Note that $h(c) = false$, because otherwise $h(S)$ would be $true$, which contradicts S's unsatisfiability. Now, suppose that an assignment h' that is different than h in only one literal $l \in c$ satisfies all the clauses in S other than exactly one clause $c' \in S$. Hence $h'(S \setminus c') = true$, which means that like c, c' must also be in any unsatisfiable subset of S, and can therefore be marked as necessary as well. Rotation flips the values of each of c's literals one at a time in search of such clauses. When one is found, rotation is called recursively with c'. This algorithm is summarized in Fig. 1(a). We observe that rotation, proposed in the context of assumption-based MUS extraction, can be integrated into our resolution-based algorithm without any changes.

Fig 1(b) shows ERMR (Eager Recursive Model Rotation) – an improvement to rotation that weakens rotation's terminating condition. The reader may benefit from first reading the main algorithm in Fig. 2(a), which calls ERMR. The only difference between ERMR and RMR is that ERMR may call rotation with a clause that is already in M, the reason being that it can lead to additional marked clauses owing to the fact that the call is with a different assignment. Clearly there is a tradeoff between the time saved by detecting more clauses for M and the time dedicated to the search. For example, one may run RMR with more than one satisfying assignment as a starting point, but this will require additional SAT calls to find extra satisfying assignments. ERMR refrains from additional SAT calls. Rather it changes the stopping criterion: instead of stopping when $c \in M$ (line 4 in Fig. 1(a)), it stops when $c \in K$, where K holds the clauses that were discovered in the *current* call from MUS. There are other variations on weakening the terminating condition of rotation in the literature [5], [6]. We leave to future study a detailed comparison of our algorithm to these works.

C. Path Strengthening

Path strengthening relies on the following property, which we call *cut falsifiability* (observed already in [12], [20]). Let S be an unsatisfiable formula, π its resolution proof, and c a candidate clause. Let ρ_c be the subgraph of π containing all the clauses that appear on at least one path from c to the

empty clause $\square$ (including c and $\square$). Then, any model h to $S \setminus \{c\}$ must falsify at least one clause in any vertex cut of ρ_c (since otherwise a satisfiable vertex cut in π would exist). An immediate corollary is that *all* the clauses in *some* path from c to $\square$ must be falsified by any model h to $S \setminus \{c\}$.

We use this property as follows. Let $P = [c_0 = c, c_1, \ldots, c_m]$ be a path in the resolution proof starting from a candidate clause c. P is the *longest unique prefix* if it is the longest path starting at c, such that each $c_i \in P$ has only one child (that is, c participates in the derivation of one clause only). *Path strengthening* is based on the following property, induced by cut falsifiability: all the clauses of P must be falsified in any model h to $S \setminus \{c\}$. Fig. 2(b) shows a variant of the main algorithm in which path strengthening has been applied: each invocation of the SAT solver is carried out under the assumptions $\neg P = \{\neg c_0, \ldots, \neg c_m\}$. Before each iteration our algorithm attempts to increase P length by removing from the resolution proof clauses that are not backward reachable from the empty clause. Note that whenever P contains clauses which do not subsume c, path strengthening will provide more assumptions to the solver than redundancy removal; hence path strengthening is expected to be more efficient than redundancy removal.

Cut falsifiability-based techniques are not immediately compliant with clause set refinement, since clause set refinement requires solving *without assumptions*. `MUSer2` solves this problem for redundancy removal by applying clause set refinement only when the assumptions are not used in the proof; otherwise it skips clause set refinement. Our path strengthening algorithm applies clause set refinement when either the assumptions are not used in the proof or whenever the N latest iterations applied path strengthening and the result was unsatisfiable, N being a user-given threshold.

III. Experimental Results

We checked the impact of our algorithms when applied to the 295 instances used for the MUS track of the SAT 2011 competition. For the experiments we used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The time-out was set to 1800 sec. The implementation was done in `HaifaMUC`. We refer to a configuration of `HaifaMUC` that implements the deletion-based algorithm with incremental SAT and clause set refinement as Base. We compare our tool to the latest version of `MUSer2` [7] and `Minisatabb` [18]. Extended experimental data is available from the second author's home page.

Fig. 3 summarizes the main results. Several observations are in order: 1) rotation is very useful; 2) eager rotation is effective; 3) optimizations **A** and **D** are useful, while optimization **B** is beneficial only if delayed until the second satisfiable iteration (2 being the optimal value, based on experiments); 4) path strengthening (with N=20, 20 being the optimal value experimentally) is more beneficial than redundancy removal, and finally 5) `HaifaMUC`, enhanced by all our algorithms, is 2.18x faster than `MUSer2` and solves 13 more instances, and is 48% faster than `Minisatabb` and solves 4 more instances. `HaifaMUC` is faster than `Minisatabb` on 196 instances, while `Minisatabb` is faster than `HaifaMUC` on 15 instances. Fig. 5 shows a cactus plot comparing Base, `MUSer2`, `Minisatabb` and the new best configuration of `HaifaMUC`, while Fig. 4 compares `HaifaMUC` to `Minisatabb`.

IV. Conclusion

We proposed a number of algorithms for speeding up MUS extraction. First, we adapted GMUS-oriented MUS-biased search algorithms to plain MUS extraction. Second, we integrated model rotation into resolution-based MUS extraction. Third, we introduced an enhancement to rotation, called eager rotation. Finally, we introduced a new enhancement, path strengthening, to resolution-based MUS extraction. We implemented the algorithms in the resolution-based MUS extractor `HaifaMUC`, which, as a result, outperformed the leading MUS extractors `MUSer2` and `Minisatabb`.

V. Acknowledgments

The authors would like to thank Daher Kaiss for supporting this work and Paul Inbar for editing the paper.

References

[1] Silva, J.P.M.: Minimal unsatisfiability: Models, algorithms and applications (invited paper). In: ISMVL'10. (2010) 9–14
[2] Nadel, A.: Boosting minimal unsatisfiable core extraction. In: FMCAD'10. (2010) 221–229
[3] Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: SAT'11. (2011) 174–187
[4] Silva, J.P.M., Lynce, I.: On improving MUS extraction algorithms. In: SAT'11. (2011) 159–173
[5] Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: CP'12. (2012) 672–687
[6] Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2) (2012) 97–116
[7] Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. JSAT **8**(1/2) (2012) 123–128
[8] Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: Preliminary Proceedings of SAT'03. (2003)
[9] Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE'03. (2003) 886–891
[10] Strichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: CHARME'01. (2001) 58–70
[11] Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. **89**(4) (2003)
[12] Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: SAT'06. (2006) 36–41
[13] Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. INFORMS Journal on Computing **3**(2) (1991) 157–168
[14] Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving over-determined constraint satisfaction problems. In: IJCAI'93. (1993) 276–281
[15] Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning **40**(1) (2008) 1–33
[16] Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: FMCAD'11. (2011) 37–40
[17] van Maaren, H., Wieringa, S.: Finding guaranteed MUSes fast. In: SAT'08. (2008) 291–304
[18] Lagniez, J.M., Biere, A.: Factoring out assumptions to speed up MUS extraction. In: SAT'13. (2013) 276–292
[19] Sörensson, N., Biere, A.: Minimizing learned clauses. In: SAT'09. (2009) 237–243
[20] Nadel, A.: Understanding and Improving a Modern SAT Solver. PhD thesis, Tel Aviv University, Tel Aviv, Israel (August 2009)

```
1: function RMR(S, M, c, h)   ▷ recursive model rotation
2:     for all x ∈ Var(S) do
3:         h' = h[x ← ¬x];          ▷ swap assignment of x
4:         if UnsatSet(S, h') = {c'} and c' ∉ M then
5:             M = M ∪ {c'};
6:             RMR (S, M, c', h');
```

(a)

```
1: function ERMR(S, M, K, c, h)   ▷ Initially K = {c}
2:     for all x ∈ Var(S) do
3:         h' = h[x ← ¬x];
4:         if UnsatSet(S, h') = {c'} and c' ∉ K then
5:             K = K ∪ {c'};
6:             if c' ∉ M then M = M ∪ {c'};
7:             ERMR (S, M, K, c', h');
```

(b)

Fig. 1. (a) The recursive model rotation of [16], where $UnsatSet(S, h')$ is the subset of S's clauses that are unsatisfied by the assignment h', and (b) our modified version. K is a set of clauses that is initialized to c before calling ERMR. $K \subseteq M$ is an invariant, and hence ERMR is called at least as many times as RMR in (a).

```
1: function MUS(unsatisfiable formula S)
2:     M = ∅;
3:     while true do
4:         choose c ∈ S \ M. If there is none, break;
5:         if SAT(S \ {c}) then
6:             K = {c};
7:             M = ERMR (S, c, M, K, h)
8:         else
9:             S = core;
```

(a)

```
1: function MUS(unsatisfiable formula S)
2:     M = ∅;
3:     while true do
4:         choose c ∈ S \ M. If there is none, break;
5:         let P be the longest unique prefix
6:         discard clauses not backward reachable from □
7:         if SAT(S \ {c}, {¬c_i | c_i ∈ P}) then
8:             K = {c}; M = ERMR (S, c, M, K, h)
9:         else
10:            if ¬P not used in proof then S = core;
11:            else
12:                S = S \ {c}
13:                if condition then   ▷ Heuristic. See text
14:                    SAT(S);         ▷ guaranteed unsat
15:                    S = core;
```

(b)

Fig. 2. (a) Deletion-based MUS extraction enhanced by eager rotation and clause set refinement, where h is the satisfying assignment, and $core$ is the unsatisfiable core (b) an improvement based on path strengthening. In line 7 the literals defined by $\{\neg c_i \mid c_i \in P\}$ are assumptions.

	Base	rot	erot	erot_AD	erot_ABD	erot_AB2D	erot_AB2CD	**erot_AB2CD_rr**	**erot_AB2CD_ps20**	MUSer2	Minisatabb
Time	93931	48018	44335	36295	37798	32968	32918	30800	**27263**	59502	40485
Unsolved	30	12	10	8	13	8	8	6	**4**	17	8

Fig. 3. Total run-time in sec. and number of unsolved instances for various solvers, when applied to the 295 instances from the 2011 MUS competition, excluding 12 instances which were not solved by any of the solvers (the time-out value of 1800 sec. was added to the run-time when a memory-out occured). Base is defined in Sect. III, rot = Base+rotation, erot = Base+eager rotation. **A**, **B**, **C**, and **D** correspond to the optimizations defined in Sect. II-A. '2' in AB2CD means that the optimization was invoked after the 2nd satisfiable result. 'rr' refers to redundancy removal combined with clause set refinement using `MUSer2`'s scheme, described in Sect. II-C. 'ps20' means that path strengthening with $N = 20$ was applied as described in Sect. II-C.

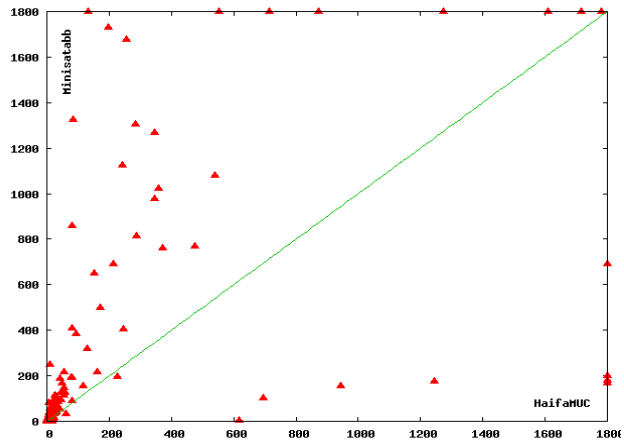


Fig. 4. Direct comparison of the new best configuration of `HaifaMUC` erot_AB2CD_ps20 (X-Axis) and `Minisatabb` (Y-Axis).

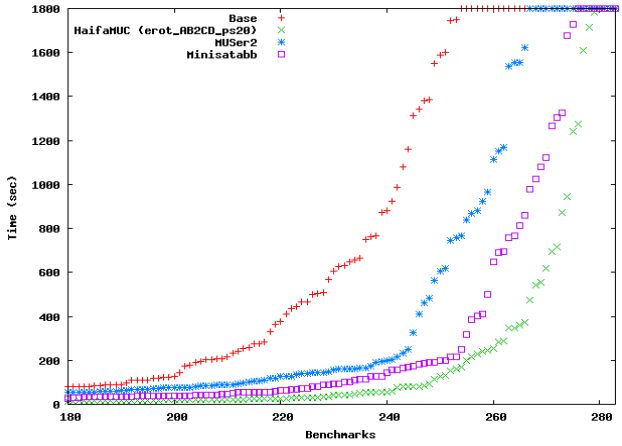


Fig. 5. Comparison of Base, `MUSer2`, `Minisatabb`, and the new best configuration of `HaifaMUC` erot_AB2CD_ps20. The graph shows the number of solved instances (X-Axis) per time-out in seconds (Y-Axis) for each solver.

Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, Josef Widder
Vienna University of Technology (TU Wien)

***Abstract*—We introduce an automated parameterized verification method for fault-tolerant distributed algorithms (FTDA). FTDAs are parameterized by both the number of processes and the assumed maximum number of faults. At the center of our technique is a parametric interval abstraction (PIA) where the interval boundaries are arithmetic expressions over parameters. Using PIA for both data abstraction and a new form of counter abstraction, we reduce the parameterized problem to finite-state model checking. We demonstrate the practical feasibility of our method by verifying safety and liveness of several fault-tolerant broadcasting algorithms, and finding counter examples in the case where there are more faults than the FTDA was designed for.**

I. Introduction

Fault-tolerant distributed algorithms (FTDA) constitute a core topic of distributed algorithm theory, with a rich body of results [27], [2]. Yet, they have not been systematically studied from a model checking point of view. For FTDAs one typically considers systems of n processes out of which at most t may be faulty. In this paper we consider various faults such as crash faults, omissions, and Byzantine faults. As FTDAs are parameterized in n and t, we require parameterized verification to establish the correctness of an FTDA. The pragmatic approach to verify a system of *fixed* size is not practical, as only very small instances can be verified due to state space explosion [24], [36], [34]. While in classic parameterized model checking the number of processes n is the sole parameter, for FTDAs, t is also a parameter, and is essentially a fraction of n, expressed by a *resilience condition*, e.g., $n > 3t$. Thus, one has to reason about all runs with $n-f$ non-faulty and f faulty processes, where $f \leq t$ and $n > 3t$.

From an operational viewpoint, FTDAs typically consist of multiple processes that communicate by passing messages. As senders can be faulty, a receiver cannot wait for a message from a specific sender process. Thus, most FTDAs use counters to reason about the environment; e.g., if a process receives a certain message from more than t distinct senders, then one of the senders must be non-faulty. A large class of FTDAs expresses these counting arguments using *threshold guards:*

```
if received <m> from t+1 distinct processes
then action(m);
```

Threshold guards generalize existential and universal guards [16], i.e., rules that wait for messages from at least one or all processes, respectively. As can be seen from the above example, and as discussed in [24], existential and universal guards are not sufficient to capture advanced FTDAs: Threshold guards are a basic building block that has been used in various environments (various degrees of synchrony, fault assumptions, etc.) and FTDAs, such as consensus [15], software and hardware clock synchronization [32], [19], approximate agreement [14], and k-set agreement [13]. The ability to efficiently reason about these guards is thus a keystone for automated parameterized verification of such algorithms.

This paper considers parameterized verification of FTDAs with threshold guards and resilience conditions. We introduce a framework based on a new form of control flow automata that captures the semantics of threshold-guarded FTDAs, and propose a novel two-step abstraction technique. It is based on *parametric interval abstraction* (PIA), a generalization of interval abstraction where the interval borders are expressions over *parameters* rather than constants. Using the PIA domain, we obtain a finite-state model checking problem in two steps:

Step 1: PIA data abstraction. We evaluate the threshold guards over the parametric intervals. Thus, we abstract away unbounded variables and parameters from the process code. We obtain a parameterized system where the replicated processes are finite-state and independent of the parameters.

Step 2: PIA counter abstraction. We use a new form of counter abstraction where the process counters are abstracted to PIA. As Step 1 guarantees that we need only finitely many counters, PIA counter abstraction yields a finite-state system.

To evaluate the precision of our abstractions, we implemented our abstraction technique in a tool chain, and conducted experiments on several FTDAs. Our experiments showed the need for abstraction refinement to deal with spurious counterexamples [7] that are due to parameterized abstraction and fairness. This required novel refinement techniques, which we also discuss in this paper. In addition to refinement of PIA counter abstraction, which is automated in a loop using a model checker and an SMT solver, we are also exploiting simple user-provided invariant candidates (as in [28], [35]) to refine the abstraction.

We verify several FTDAs that have been derived from the well-known distributed broadcast algorithm by Srikanth and Toueg [32], [33], and a folklore reliable broadcasting algorithm [2, Sect. 8.2.5.1]. Each of these FTDAs tolerates different faults (e.g., crash, omission, Byzantine), and uses different threshold guards. To the best of our knowledge, we are the first to achieve parameterized automated verification of Byzantine FTDAs.

Supported by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) grants PROSEED, ICT12-059, and VRG11-005. Details that had to be omitted from this paper can be found in [23].

ISBN 978-0-9835678-3-7/13.

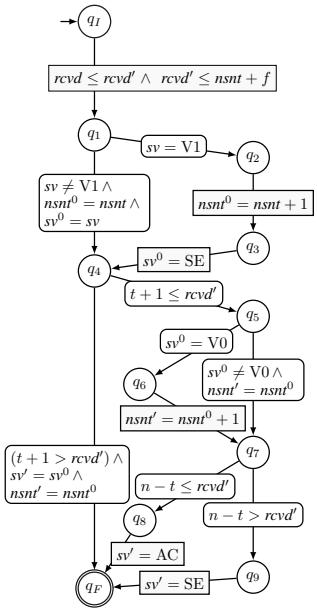


Fig. 1. CFA of our case study for Byzantine faults.

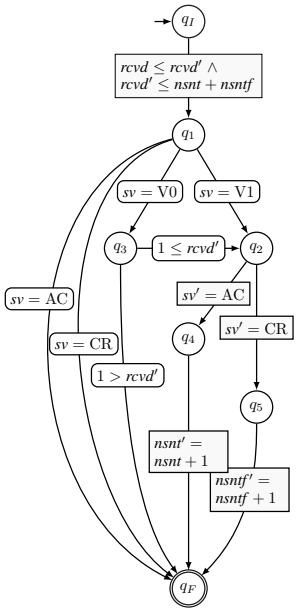


Fig. 2. CFA of FTDA from [18] (if x' is not assigned, then $x' = x$).

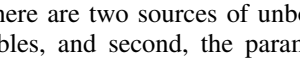
II. Our Approach at a Glance

To give an intuition of our method, we start with the control flow automaton (CFA) given in Figure 1 that formalizes our case study FTDA. The CFA uses the shared integer variable *nsnt* (capturing the number of messages sent by non-faulty processes), the local integer variable *rcvd* (storing the number of messages received by the process so far), and the local status variable *sv*, which ranges over a finite domain (capturing the local progress w.r.t. the FTDA). In [24] we show that this formalization captures the logic of our case study FTDA.

We use the CFA to represent one atomic *step* of the FTDA: Each edge is labeled with a guard. A path from q_I to q_F induces a conjunction of all the guards along it, and imposes constraints on the variables before the step (e.g., *sv*), after the step (*sv*′), and temporary variables (sv^0). If one fixes the variables before the step, different valuations (of the primed variables) that satisfy the constraints capture non-determinism.

A system consists of $n - f$ processes that concurrently execute the code corresponding to the CFA, and communicate via *nsnt*. Thus, there are two sources of unboundedness: first, the integer variables, and second, the parametric number of processes. We deal with these two issues in two steps.

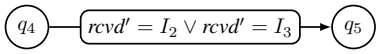
Step 1: PIA data abstraction. We observe that the CFA contains several transitions which are labeled with *threshold guards* that refer to (unbounded) variables and parameters. For instance, the CFA in Figure 1 contains the following transition, which is labeled with a threshold guard:



The CFA also contains a guard $n - t \leq rcvd'$. Actually, the correctness of the FTDA is based on the fact that the values of the thresholds, e.g., $t+1$ and $n-t$, are sufficiently far apart from each other under the resilience condition $n > 3t \wedge f \leq t$; in particular, $(n - t) - f \geq t + 1$. These properties are also used in the manual proofs [33]. We observe that such FTDAs are designed by carefully choosing the thresholds and the resilience condition. Consequently, our abstraction must be sufficiently precise to preserve the relationship between thresholds and the resilience condition.

The second important observation is that it is not necessary to keep track of the precise value of variables that are compared against thresholds, e.g., *rcvd*′. Rather, in our case study, it is sufficient to know whether *rcvd*′ lies in the interval $[0, t + 1[$, or $[t + 1, n - t[$, or $[n - t, \infty[$, in order to determine which of the threshold guards of the CFA are satisfied. Our *parametric interval abstraction* PIA exploits this idea. In addition, in Step 2 we will see that we also have to distinguish 0 from other values. Thus, PIA consists of mapping integers to a finite domain of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t + 1[$ and $I_2 = [t + 1, n - t[$ and $I_3 = [n - t; \infty[$.

Then, we replace the guards that refer to unbounded variables and parameters by their existential abstraction. For instance, the above transition with the guard "$t + 1 \leq rcvd'$" means that *rcvd*′ lies in the intervals $[t+1, n-t[$ or $[n-t, \infty[$. As these correspond to the abstract intervals I_2 and I_3, respectively, we can replace the guard by:



The abstraction of the guard "$nsnt^0 = nsnt + 1$" can be expressed similarly, as later discussed in Figure 4. The expression "$rcvd' \leq nsnt + f$", which is also used in a guard, is more complicated as it involves two variables and a parameter. Still, the basic abstraction idea is the same. The corresponding abstract expression has the form $(rcvd' = I_0 \wedge nsnt = I_0) \vee (rcvd' = I_0 \wedge nsnt = I_1) \vee \cdots \vee (rcvd' = I_3 \wedge nsnt = I_3)$.

These abstract guards are Boolean expressions over equalities between variables and abstract values. Therefore, it is sufficient to interpret the variables *nsnt* and *rcvd* over the finite domain. Hence, all variables range over finite domains, and we arrive at finite state processes in this way. Our system, however, is still parameterized, namely, in the number of processes.

Step 2: PIA counter abstraction. We reduce this system to a finite state system using the following two ideas. First, we change to a counter-based representation, i.e., the global state is represented by the (abstract) shared variable *nsnt*, and by one counter for each of the local states. A counter stores how many processes are in the corresponding local state. Second, as processes interact only via the *nsnt* variable, precisely counting processes in certain states may not be necessary; as *nsnt* already ranges over the abstract domain, it is natural to count processes in terms of the same abstract domain.

The local state of a process is determined by the values of *sv* and *rcvd*. Thus, we denote by $\kappa[x, y] = I$ that the number of processes with $sv = x$ and $rcvd = y$ lies in the abstract interval I. Then, in Figure 3, the state s_0 represents the initial states with $t + 1$ to $n - t - 1$ processes having $sv = \mathtt{V0}$ and 1 to t processes having $sv = \mathtt{V1}$. (We omit local states that have the counter value I_0 to facilitate reading.)

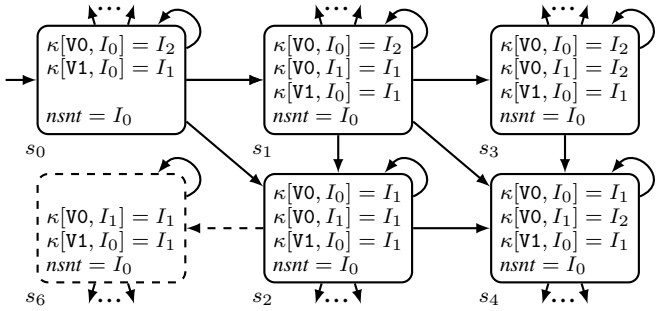


Fig. 3. A small part of the transition system obtained by counter abstraction. As shown by our experimental data in Table I of Section VII, the reachable state space is substantially larger.

Figure 3 gives a small part of the transition system obtained from the counter abstraction starting from initial state s_0. Each transition corresponds to one process taking a step in the concrete system. For instance, in the transition (s_0, s_2) a process with local state $[\texttt{V0}, I_0]$ changes its state to $[\texttt{V0}, I_1]$. Therefore, the counter $\kappa[\texttt{V0}, I_0]$ is decremented and the counter $\kappa[\texttt{V0}, I_1]$ is incremented. However, as we interpret counters over the abstract domain, the operations of incrementing and decrementing a counter are actually non-deterministic. Consequently, the transition (s_0, s_1) captures the same concrete local step as (s_0, s_2). In (s_0, s_1), the non-deterministic decrement of the abstract counter $\kappa[\texttt{V0}, I_0]$ did not change its value.

Typically, the specifications of FTDAs refer to global states where "there is a process in a given local state" or "all processes are in a given local state." To express this via counters, we have to check whether counter values are I_0.

Abstraction refinement. Our abstraction steps result in a system which is an over-approximation of all systems with fixed parameters. For instance, the non-determinism in the counters may "increase" or "decrease" the number of processes in a system, although in all concrete system the number of processes is constant: Consider the transition (s_2, s_6) in Figure 3, and let x, y, z be the non-negative integers that are in s_2 abstracted to $\kappa[\texttt{V0}, I_0]$, $\kappa[\texttt{V0}, I_1]$, and $\kappa[\texttt{V1}, I_0]$, respectively. Similarly y' and z' are abstracted to $\kappa[\texttt{V0}, I_1]$ and $\kappa[\texttt{V1}, I_0]$ in s_6. If the following inequalities do not have a solution under the resilience condition $(n > 3t, t \geq f)$, then there is no concrete system with a transition between two states that are abstracted to s_2 and s_6, respectively.

$$1 \leq x < t+1,\ 1 \leq y < t+1,\ 1 \leq z < t+1,$$
$$1 \leq y' < t+1,\ 1 \leq z' < t+1,$$
$$x + y + z = y' + z' = n - f.$$

We use an SMT solver for this, and examine each transition of a counterexample returned by a model checker. If a transition is spurious, then we remove it from the abstract system.

Related abstractions. Interval abstraction [10] is a natural solution to the problem of unboundedness of local variables. However, if we fixed the interval bounds to numeric values, then they would not be aligned to the thresholds, and the abstraction would not be sufficiently precise to do parametric verification. At the same time, we do not have to deal with symbolic ranges over *variables* in the sense of [30], because for FTDAs the interval bounds are *constant* in each run.

Further, we want to produce a single process skeleton that is independent of parameters and captures the behavior of *all* process instances. This can be done by using ideas from existential abstraction [9], [12], [25] and sound abstraction of fairness constraints [25]. We combine these two ideas to arrive at PIA data abstraction.

The PIA counter abstraction is similar to [29], in that counters range over an abstract domain, and increment and decrement is done using existential abstraction. The domain in [29] consists of three values representing 0, 1, or *more*. This domain is sufficient for mutual-exclusion-like problems: It allows to distinguish good from bad states, while it is not possible (and also not necessary) to distinguish two bad states: A bad state is one where at least two processes are in the critical section, which is precisely abstracted in the three-valued domain. However, two bad states where, e.g., 2 and 3 processes are in the critical section, respectively, cannot be distinguished. Verification of threshold-based FTDAs requires more involved counting; e.g., we have to capture whether at least $n - t$ processes or at most t processes incremented *nsnt*. Therefore, we use counters from the PIA domain.

III. System Model with Multiple Parameters

In this section we develop all notions that are required to precisely state the parameterized model checking problem for multiple parameters. As running example, we use the parameters mentioned above, namely, the number of processes n, the upper bound on the number of faults t, and the actual number of faults f. We start to define parameterized processes (that access shared variables) in a way that allows us to modularly compose them into a parameterized system instance.

We apply this modeling to verify FTDAs as follows: as input we take a process description that uses the parameters n and t in the code. From this we construct a system instance parameterized with n, t, and f, which then describes all runs of an algorithm in which exactly f faults occur. The verification problem for a distributed algorithm in the *concrete case* with fixed n and t is the composition of model checking problems that differ in the actual value of $f \leq t$. This modeling also allows us to set $f = t + 1$, which models runs in which more faults occur than expected, and search for counterexamples. For the *parameterized case*, we introduce a resilience condition on these parameters, and require to verify the algorithm for all values of parameters that satisfy the resilience condition.

We define the parameters, local variables of the processes, and shared variables referring to a single *domain* D that is totally ordered and has the operations of addition and subtraction. In this paper we assume that D is the set of nonnegative integers $\mathbb{N}_0$.

We start with some notation. Let Y be a finite set of variables ranging over D. We denote by $D^{|Y|}$, the set of all $|Y|$-tuples of variable values. Given $\mathbf{s} \in D^{|Y|}$, we use the expression $\mathbf{s}.y$, to refer to the value of a variable $y \in Y$ in

vector $\mathbf{s}$. For two vectors $\mathbf{s}$ and $\mathbf{s}'$, by $\mathbf{s} =_X \mathbf{s}'$ we denote the fact that for all $x \in X$, $\mathbf{s}.x = \mathbf{s}'.x$ holds.

Process. The set of variables V is $\{sv\} \cup \Lambda \cup \Gamma \cup \Pi$: The variable sv is the *status variable* that ranges over a finite set *SV* of *status values*. The finite set Λ contains variables that range over the domain D. The variable sv and the variables from Λ are *local variables*. The finite set Γ contains the *shared variables* that range over D. The finite set Π is a set of *parameter variables* that range over D, and the *resilience condition RC* is a predicate over $D^{|\Pi|}$. In our example, $\Pi = \{n, t, f\}$, and the resilience condition $RC(n,t,f)$ is $n > 3t \ \wedge \ f \leq t \ \wedge \ t > 0$. Then, we denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in D^{|\Pi|} \mid RC(\mathbf{p})\}$.

A process operates on states from the set $S = SV \times D^{|\Lambda|} \times D^{|\Gamma|} \times D^{|\Pi|}$. Each process starts its computation in an initial state from a set $S^0 \subseteq S$. A relation $R \subseteq S \times S$ defines *transitions* from one state to another, with the restriction that the values of parameters remain unchanged, i.e., for all $(\mathbf{s}, \mathbf{t}) \in R$, $\mathbf{s} =_\Pi \mathbf{t}$. Then, a *parameterized process skeleton* is a tuple $\mathsf{Sk} = (S, S^0, R)$.

We get a process instance by fixing the parameter values $\mathbf{p} \in D^{|\Pi|}$: one can restrict the set of process states to $S|_{\mathbf{p}} = \{\mathbf{s} \in S \mid \mathbf{s} =_\Pi \mathbf{p}\}$ as well as the set of transitions to $R|_{\mathbf{p}} = R \cap (S|_{\mathbf{p}} \times S|_{\mathbf{p}})$. Then, a *process instance* is a process skeleton $\mathsf{Sk}|_{\mathbf{p}} = (S|_{\mathbf{p}}, S^0|_{\mathbf{p}}, R|_{\mathbf{p}})$ where $\mathbf{p}$ is constant.

System Instance. For fixed admissible parameters $\mathbf{p}$, a distributed system is modeled as an asynchronous parallel composition of identical processes $\mathsf{Sk}|_{\mathbf{p}}$. The number of processes depends on the parameters. To formalize this, we define the size of a system (the number of processes) using a function $N \colon \mathbf{P}_{RC} \to \mathbb{N}_0$, for instance, when modeling only correct processes explicitly, we use $n - f$ for $N(n,t,f)$.

Given $\mathbf{p} \in \mathbf{P}_{RC}$, and a process skeleton $\mathsf{Sk} = (S, S^0, R)$, a system instance is defined as an asynchronous parallel composition of $N(\mathbf{p})$ process instances, indexed by $i \in \{1, \ldots, N(\mathbf{p})\}$, with standard interleaving semantics. Let AP be a set of atomic propositions. A *system instance* $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a Kripke structure $(S_I, S_I^0, R_I, \mathrm{AP}, \lambda_I)$ where:

- $S_I = \{(\sigma[1], \ldots, \sigma[N(\mathbf{p})]) \in (S|_{\mathbf{p}})^{N(\mathbf{p})} \mid \forall i, j \in \{1, \ldots, N(\mathbf{p})\}, \sigma[i] =_{\Gamma \cup \Pi} \sigma[j]\}$ is the set of *(global) states*. Informally, a global state σ is a Cartesian product of the state $\sigma[i]$ of each process i, with identical values of parameters and shared variables at each process.
- $S_I^0 = (S^0)^{N(\mathbf{p})} \cap S_I$ is the set of *initial (global) states*, where $(S^0)^{N(\mathbf{p})}$ is the Cartesian product of initial states of individual processes.
- A transition (σ, σ') from a global state $\sigma \in S_I$ to a global state $\sigma' \in S_I$ belongs to R_I iff there is an index i, $1 \leq i \leq N(\mathbf{p})$, such that:
  - (MOVE) The i-th process *moves*: $(\sigma[i], \sigma'[i]) \in R|_{\mathbf{p}}$.
  - (FRAME) The values of the local variables of the other processes are preserved: for every process index $j \neq i$, $1 \leq j \leq N(\mathbf{p})$, it holds that $\sigma[j] =_{\{sv\} \cup \Lambda} \sigma'[j]$.
- $\lambda_I : S_I \to 2^{\mathrm{AP}}$ is a state labeling function.

Remark 1: The set of global states S_I and the transition relation R_I are preserved under every transposition $i \leftrightarrow j$ of process indices i and j in $\{1, \ldots, N(\mathbf{p})\}$. That is, every system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *fully symmetric* by construction.

Atomic Propositions. We define the set of atomic propositions AP to be the disjoint union of AP_{SV} and AP_D: The set AP_{SV} contains propositions that capture comparison against a given status value $Z \in SV$, i.e., $[\forall i.\ sv_i = Z]$ and $[\exists i.\ sv_i = Z]$. Further, the set of atomic propositions AP_D captures comparison of variables x, y, and a linear combination c of parameters from Π; AP_D consists of propositions of the form $[\exists i.\ x_i + c < y_i]$ and $[\forall i.\ x_i + c \geq y_i]$.

The labeling function λ_I of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ maps a state σ to expressions p from AP as follows (the existential case is defined accordingly using disjunctions):

$$[\forall i.\ sv_i = Z] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].sv = Z)$$

$$[\forall i.\ x_i + c \geq y_i] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].x + c(\mathbf{p}) \geq \sigma[i].y)$$

Temporal Logic. We specify properties using temporal logic $\mathsf{LTL}_{\setminus \mathsf{X}}$ over AP_{SV}. We use the standard definitions of paths and $\mathsf{LTL}_{\setminus \mathsf{X}}$ semantics [6]. A formula of $\mathsf{LTL}_{\setminus \mathsf{X}}$ is defined inductively as: (i) a literal p or $\neg p$, where $p \in \mathrm{AP}_{SV}$, or (ii) $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi \vee \psi$, and $\varphi \wedge \psi$, where φ and ψ are $\mathsf{LTL}_{\setminus \mathsf{X}}$ formulas.

Fairness. We are interested in verifying safety and liveness properties. The latter can be usually proven only in the presence of fairness constraints. As in [25], [29], we consider verification of safety and liveness in systems with *justice* fairness constraints. We define fair paths of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ using a set of justice constraints $J \subseteq \mathrm{AP}_D$. A path π of a system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *J-fair* iff for every $p \in J$ there are infinitely many states σ in π with $p \in \lambda_I(\sigma)$. By $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$ we denote that the formula φ holds on all J-fair paths of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$.

Definition 2: Given a system description containing

- a domain D,
- a parameterized process skeleton $\mathsf{Sk} = (S, S_0, R)$,
- a resilience condition *RC* (generating a set of admissible parameters $\mathbf{P}_{RC}$),
- a system size function N,
- justice requirements J,

and an $\mathsf{LTL}_{\setminus \mathsf{X}}$ formula φ, the *parameterized model checking problem* (PMCP) is to verify $\forall \mathbf{p} \in \mathbf{P}_{RC}.\ \mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$.

IV. Threshold-guarded FTDAs

In [24], we formalized threshold-guarded FTDAs in Promela. In order to introduce our abstraction technique, we propose a language-independent approach that focuses on the control flow and is based on control flow automata (CFA) [21].

A *guarded control flow automaton* (CFA) is an edge-labeled directed acyclic graph $\mathcal{A} = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. A path from q_I to q_F is used to describe *one step* of a distributed algorithm. The edges have the form

$E \subseteq Q \times \texttt{guard} \times Q$, where guard is defined as an expression of one of the following forms where $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$:

- if $Z \in SV$, then $sv = Z$ and $sv \neq Z$ are *status guards*;
- if x is a variable in D and $\lhd \in \{\leq, >\}$, then

$$a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i \ \lhd \ x$$

  is a *threshold guard*;
- if $y, z_1, \ldots, z_k$ are variables in D for $k \geq 1$, and $\lhd \in \{=, \neq, <, \leq, >, \geq\}$, and $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, then

$$y \ \lhd \ z_1 + \cdots + z_k + \big(a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i\big)$$

  is a *comparison guard*;
- a conjunction $g_1 \wedge g_2$ of guards g_1 and g_2 is a guard.

Status guards are used to capture the basic control flow. Threshold guards capture the core primitive of the FTDAs we consider. Finally, comparison guards are used to model send and receive operations. Figure 1 shows an example CFA with $\Gamma = \{nsnt\}$, $\Lambda = \{rcvd\}$, and $\Pi = \{n, t, f\}$.

Obtaining a Skeleton from a CFA. One step of a process skeleton is defined by a path from q_I to q_F in a CFA. Given *SV*, Λ, Γ, Π, *RC*, and a CFA $\mathcal{A}$, we define the process skeleton $\mathsf{Sk}(\mathcal{A}) = (S, S^0, R)$ induced by $\mathcal{A}$ as follows: The set of variables used by the CFA is $W \supseteq \Pi \cup \Lambda \cup \Gamma \cup \{sv\} \cup \{x' \mid x \in \Lambda \cup \Gamma \cup \{sv\}\}$, which may contain also temporary variables. A variable x corresponds to the value before a step, x' to the value after the step, and $x^0, x^1, \ldots$ to intermediate values. A path p from q_I to q_F induces a conjuction of all the guards along it. We call a mapping v from W to the values from the respective domains a *valuation*. We write $v \models p$ to denote that the valuation v satisfies the guards of the path p. We define the mapping between a CFA $\mathcal{A}$ and the transition relation of a process skeleton $\mathsf{Sk}(\mathcal{A})$: If there is a path p and a valuation v with $v \models p$, then v defines a single transition (s, t) of a process skeleton $\mathsf{Sk}(\mathcal{A})$, if for each variable $x \in \Lambda \cup \Gamma \cup \{sv\}$ it holds that $s.x = v(x)$ and $t.x = v(x')$ and for each parameter variable $z \in \Pi$, $s.z = t.z = v(z)$. Finally, the initial states S^0 need to be specified. For the type of algorithms we consider in this paper, all variables of the skeleton that range over D are initialized to 0, and *sv* ranging over *SV* takes an initial value from a fixed subset of *SV*. (For other algorithms, or self-stabilizing systems, one would choose different initializations.)

Remark 3: It might seem restrictive that our guards do not contain, e.g., increment, assignments, non-deterministic choice from a range of values. However, all these statements can be translated in our form using the SSA transformation algorithm from [11]. For instance, Figure 1 has been obtained from the Promela case study in [24], which contains the mentioned statements. Figures 1 and 2 provide two of the algorithms we have used for our experiments in Section VII.

Definition 4 (PMCP for CFA): We define the Parameterized Model Checking Problem for CFA $\mathcal{A}$ by specializing Definition 2 to the parameterized process skeleton $\mathsf{Sk}(\mathcal{A})$.

The problem given in Definition 4 is undecidable even if the CFA contains only status variables [23].

V. Abstraction Scheme

The input to our abstraction method is the infinite parameterized family $\mathcal{F} = \{\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of Kripke structures specified via a CFA $\mathcal{A}$. The family $\mathcal{F}$ has two principal sources of unboundedness: unbounded variables in the process skeleton $\mathsf{Sk}(\mathcal{A})$, and the unbounded number of processes $N(\mathbf{p})$. We deal with these two aspects separately, using two abstraction steps, namely the *PIA data abstraction* and the *PIA counter abstraction*. In both abstraction steps we use the parametric interval abstraction PIA.

Given a CFA $\mathcal{A}$, let $\mathcal{G}_\mathcal{A}$ be the set of all linear combinations $a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i$ in the left-hand sides of $\mathcal{A}$'s threshold guards. Every expression ε of $\mathcal{G}_\mathcal{A}$ defines a function $f_\varepsilon \colon \mathbf{P}_{RC} \to D$. Let $\mathcal{T} = \{0, 1\} \cup \{f_\varepsilon \mid \varepsilon \in \mathcal{G}_\mathcal{A}\}$ be a finite *threshold set*, and $\mu + 1$ its cardinality. For convenience, we name elements of $\mathcal{T}$ as $\theta_0, \theta_1, \ldots, \theta_\mu$ with θ_0 corresponding to the constant function 0, and θ_1 corresponding to the constant 1. E.g., the CFA in Fig. 1 has the threshold set $\{\theta_0, \theta_1, \theta_2, \theta_3\}$, where $\theta_2(n, t, f) = t + 1$ and $\theta_3(n, t, f) = n - t$. Then, we define the domain of parametric intervals as:

$$\widehat{D} = \{I_j \mid 0 \leq j \leq \mu\}$$

Our abstraction rests on an implicit property of many FTDAs, namely, that the resilience condition *RC* induces an order on the thresholds used in the algorithm (e.g., $t+1 < n-t$).

Definition 5: The finite set $\mathcal{T}$ is uniformly ordered if for all $\mathbf{p} \in \mathbf{P}_{RC}$, and all $\theta_j(\mathbf{p})$ and $\theta_k(\mathbf{p})$ in $\mathcal{T}$ with $0 \leq j < k \leq \mu$, it holds that $\theta_j(\mathbf{p}) < \theta_k(\mathbf{p})$.

Assuming such an order does not limit the application of our approach: In cases where only a partial order is induced by *RC*, one can simply enumerate all finitely many total orders. As parameters, and thus thresholds, are kept unchanged in a run, one can verify an algorithm for each threshold order separately, and then combine the results.

Definition 5 allows us to properly define the *parameterized abstraction function* $\alpha_\mathbf{p} \colon D \to \widehat{D}$ and the *parameterized concretization function* $\gamma_\mathbf{p} \colon \widehat{D} \to 2^D$.

$$\alpha_\mathbf{p}(x) = \begin{cases} I_j & \text{if } x \in [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[\text{ for some } 0 \leq j < \mu \\ I_\mu & \text{otherwise.} \end{cases}$$

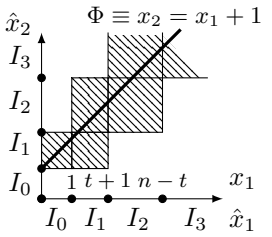
$$\gamma_\mathbf{p}(I_j) = \begin{cases} [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[& \text{if } j < \mu \\ [\theta_\mu(\mathbf{p}), \infty[& \text{otherwise.} \end{cases}$$

From $\theta_0(\mathbf{p}) = 0$ and $\theta_1(\mathbf{p}) = 1$, it immediately follows that for all $\mathbf{p} \in \mathbf{P}_{RC}$, we have $\alpha_\mathbf{p}(0) = I_0$, $\alpha_\mathbf{p}(1) = I_1$, and $\gamma_\mathbf{p}(I_0) = \{0\}$. Moreover, from the definitions of α, γ, and Definition 5 one immediately obtains:

Proposition 6: For all $\mathbf{p}$ in $\mathbf{P}_{RC}$, and for all a in D, it holds that $a \in \gamma_\mathbf{p}(\alpha_\mathbf{p}(a))$.

Definition 7: We define comparison between parametric intervals I_k and I_ℓ as $I_k \leq I_\ell$ iff $k \leq \ell$.

The PIA domain has similarities to predicate abstraction since the interval borders are naturally expressed as predicates, and computations over PIA are directly reduced to SMT solvers. However, notions such as the order of Definition 7 are not naturally expressed in terms of predicate abstraction.



$$\begin{aligned}\hat{\Phi} \equiv\ & \hat{x}_1 = I_0 \wedge \hat{x}_2 = I_1 \\ \vee\ & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_1 \\ \vee\ & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_2 \\ \vee\ & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_2 \\ \vee\ & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_3 \\ \vee\ & \hat{x}_1 = I_3 \wedge \hat{x}_2 = I_3\end{aligned}$$

Fig. 4. The shaded area approximates the line $x_2 = x_1 + 1$ along the boundaries of our parametric intervals. Each shaded rectangle corresponds to one conjunctive clause in the formula to the right. Thus, given $\Phi \equiv x_2 = x_1 + 1$, the shaded rectangles correspond to $||\Phi||_\exists$, from which we immediately construct the existential abstraction $\hat{\Phi}$.

A. PIA data abstraction

We now discuss an existential abstraction of a formula Φ that is either a threshold or a comparison guard (we consider other guards later). To this end, we introduce notation for sets of vectors satisfying Φ. According to Section IV, formula Φ has two kinds of free variables: parameter variables from Π and data variables from $\Lambda \cup \Gamma$. Let $\mathbf{x}^p$ be a vector of parameter variables $(x^p_1, \ldots, x^p_{|\Pi|})$ and $\mathbf{x}^v$ be a vector of variables $(x^v_1, \ldots, x^v_k)$ over D^k. Given a k-dimensional vector $\mathbf{d}$ of values from D, by

$$\mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi$$

we denote that Φ is satisfied on concrete values $x^v_1 = d_1, \ldots, x^v_k = d_k$ and parameter values $\mathbf{p}$. Then, we define:

$$\begin{aligned}||\Phi||_\exists = \{\hat{\mathbf{d}} \in \widehat{D}^k \mid\ & \exists \mathbf{p} \in \mathbf{P}_{RC}\ \exists \mathbf{d} = (d_1, \ldots, d_k) \in D^k. \\ & \hat{\mathbf{d}} = (\alpha_{\mathbf{p}}(d_1), \ldots, \alpha_{\mathbf{p}}(d_k)) \ \wedge\ \mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi\}\end{aligned}$$

Hence, the set $||\Phi||_\exists$ contains all vectors of abstract values that correspond to some concrete values satisfying Φ. Parameters do not appear anymore due to existential quantification. A PIA *existential abstraction* of Φ is defined to be a formula $\hat{\Phi}$ over a vector of variables $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_k)$ over $\widehat{D}^k$ such that $\{\hat{\mathbf{d}} \in \widehat{D}^k \mid \hat{\mathbf{x}} = \hat{\mathbf{d}} \models \hat{\Phi}\} \supseteq ||\Phi||_\exists$.

Computing PIA abstractions. The central property of our abstract domain is that it allows to abstract comparisons against thresholds (i.e., threshold guards) in a precise way. That is, we can abstract formulas of the form $\theta_j(\mathbf{p}) \leq x_1$ by $I_j \leq \hat{x}_1$ and $\theta_j(\mathbf{p}) > x_1$ by $I_j > \hat{x}_1$. In fact, this abstraction is precise in the following sense.

Proposition 8: For all $\mathbf{p} \in \mathbf{P}_{RC}$ and all $a \in D$:
$\theta_j(\mathbf{p}) \leq a$ iff $I_j \leq \alpha_{\mathbf{p}}(a)$, and $\theta_j(\mathbf{p}) > a$ iff $I_j > \alpha_{\mathbf{p}}(a)$.

For comparison guards we use the general form, well-known from the literature, from the following proposition.

Proposition 9: If Φ is a formula over variables $x_1, \ldots, x_k$ over D, then $\bigvee_{(\hat{d}_1, \ldots, \hat{d}_k) \in ||\Phi||_\exists} \hat{x}_1 = \hat{d}_1 \wedge \cdots \wedge \hat{x}_k = \hat{d}_k$ is a PIA existential abstraction.

If the domain $\widehat{D}$ is small (as it is in our case), then one can enumerate all vectors of abstract values in $\widehat{D}^k$ and check which belong to our abstraction $||\Phi||_\exists$, using an SMT solver. As example consider the PIA domain $\{I_0, I_1, I_2, I_3\}$ for the CFA from Fig. 1. Fig. 4 illustrates $||\Phi||_\exists$ of $x_2 = x_1 + 1$ and the use of the formula from Proposition 9.

Transforming CFA. We now describe a general method to abstract `guard` formulas, and thus construct an abstract process skeleton. To this end, we denote by α_E a mapping from a concrete formula Φ to some existential abstraction of Φ (not necessarily constructed as above). By fixing α_E, we can define an abstraction of a `guard` of a CFA:

$$abs(g) = \begin{cases} \alpha_E(g) & \text{if } g \text{ is a threshold guard} \\ \alpha_E(g) & \text{if } g \text{ is a comparison guard} \\ g & \text{if } g \text{ is a status guard} \\ abs(g_1) \wedge abs(g_2) & \text{otherwise, i.e., } g \text{ is } g_1 \wedge g_2 \end{cases}$$

By abusing the notation, for a CFA $\mathcal{A}$ by $abs(\mathcal{A})$ we denote the CFA that is obtained from $\mathcal{A}$ by replacing every guard g with $abs(g)$. Note that $abs(\mathcal{A})$ contains only guards over sv and over abstract variables over $\widehat{D}$. For model checking, we have to reason about the Kripke structures that are built using the skeletons obtained from CFAs. We denote by $\mathsf{Sk}_{abs}(\mathcal{A})$, the process skeleton that is induced by CFA $abs(\mathcal{A})$, and by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$ an instance constructed from $\mathsf{Sk}_{abs}(\mathcal{A})$.

Soundness. It can be shown that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is simulated by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$, with respect to AP_{SV}. Moreover, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.

B. PIA counter abstraction

In this section, we present a counter abstraction inspired by [29], which maps a system instance composed of *identical finite state* process skeletons to a single finite state system. We use the PIA domain $\widehat{D}$ along with abstractions $\alpha_E(\{x' = x + 1\})$ and $\alpha_E(\{x' = x - 1\})$ for the counters.

Let us consider a process skeleton $\mathsf{Sk} = (S, S_0, R)$, where $S = SV \times \tilde{D}^{|\Lambda|} \times \tilde{D}^{|\Gamma|} \times \tilde{D}^{|\Pi|}$ that is defined using an arbitrary finite domain $\tilde{D}$. We present counter abstraction over the abstract domain $\widehat{D}$ in two stages, where the first stage is only a change in representation, but not an abstraction.

Stage 1: Vector Addition System with States (VASS). Let $L = \{\ell \in SV \times \tilde{D}^{|\Lambda|} \mid \exists s \in S.\ \ell =_{\{sv\} \cup \Lambda} s\}$ be the set of *local states* of a process skeleton. As the domain $\tilde{D}$ and the set of local variables Λ are finite, L is finite. We write the elements of L as $\ell_1, \ldots, \ell_{|L|}$. We define the counting function $K \colon S_I \times L \to D$ such that $K[\sigma, \ell]$ is the number of processes i whose local state is ℓ in global state $\sigma \in S_I$, i.e., $\sigma[i] =_{\{sv\} \cup \Lambda} \ell$. Thus, we represent the system state σ as a tuple $(g_1, \ldots, g_k, K[\sigma, \ell_1], \ldots, K[\sigma, \ell_{|L|}])$, i.e., by the shared global state and by the counters for the local states. If a process moves from local state ℓ_i to local state ℓ_j, the counters of ℓ_i and ℓ_j will decrement and increment, respectively.

Stage 2: Abstraction of VASS. We abstract the counters K of the VASS representation using the PIA domain to obtain a finite state Kripke structure $\mathsf{C}(\mathsf{Sk})$. To compute $\mathsf{C}(\mathsf{Sk}) = (S_\mathsf{C}, S^0_\mathsf{C}, R_\mathsf{C}, \mathrm{AP}, \lambda_\mathsf{C})$ we proceed as follows:

A state $w \in S_\mathsf{C}$ is given by values of shared variables from the set Γ, ranging over $\tilde{D}^{|\Gamma|}$, and by a vector

$(\kappa[\ell_1], \ldots, \kappa[\ell_{|L|}])$ over the abstract domain $\widehat{D}$ from Section V. More concisely, $S_{\mathsf{C}} = \widehat{D}^{|L|} \times \tilde{D}^{|\Gamma|}$.

Definition 10: The parameterized abstraction mapping $\bar{h}^{cnt}_{\mathbf{p}}$ maps a global state σ of the system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ to a state w of the abstraction $\mathsf{C}(\mathsf{Sk})$ such that: For all $\ell \in L$ it holds that $w.\kappa[\ell] = \alpha_{\mathbf{p}}(K[\sigma, \ell])$, and $w =_\Gamma \sigma$.

From the definition, one can see how to construct the initial states. Informally, we require (1) that the initial shared states of $\mathsf{C}(\mathsf{Sk})$ correspond to initial shared states of Sk, (2) that there are actually $N(\mathbf{p})$ processes in the system, and (3) that initially all processes are in an initial state.

The intuition for the construction of the transition relation is as follows: Like in VASS, a step that brings a process from local state ℓ_i to ℓ_j can be modeled by decrementing the (non-zero) counter of ℓ_i and incrementing the counter of ℓ_j using the existential abstraction $\alpha_E(\{\kappa'[\ell_i] = \kappa[\ell_i] - 1\})$ and $\alpha_E(\{\kappa'[\ell_j] = \kappa[\ell_j] + 1\})$.

Soundness. We show that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all finite state process skeletons Sk, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is simulated by $\mathsf{C}(\mathsf{Sk})$, w.r.t. AP_{SV}. Further, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a J-fair path of $\mathsf{C}(\mathsf{Sk})$.

Theorem 11 (Soundness of data & counter abstraction): For all CFA $\mathcal{A}$, and for all formulas φ from $\mathsf{LTL}_{\setminus \mathsf{X}}$ over AP_{SV} and justice constraints $J \subseteq \mathrm{AP}_D$: if $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A})) \models_J \varphi$, then for all $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \models_J \varphi$.

VI. Abstraction Refinement

The states of the abstract system are determined by variables over $\widehat{D}$. Proposition 8 shows that we precisely abstract the relevant properties of our variables, i.e., comparisons to thresholds. Hence, the classic CEGAR approach [7], which consists of refining the state space, does not appear suitable. However, the non-determinism due to our existential abstraction leads to *spurious transitions* that one can eliminate.

We encountered two sources of spurious transitions: As discussed in Section II, transitions can "lose processes," i.e., any concretization of the abstract number of processes is less than the number of processes we started with. This is not within the assumption of FTDAs, and thus spurious. Second, in our case study (cf. Figure 1) processes increase the global variable *nsnt* by one, when they transfer to a state where the value of the status variable is in $\{\mathrm{SE}, \mathrm{AC}\}$. Hence, in concrete system instances, *nsnt* should always be equal to the number of processes whose status variable value is in $\{\mathrm{SE}, \mathrm{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system.

The experiments show that in our case studies neither losing processes nor losing messages has influence on the verification of safety specifications. However, these behaviors pose challenges for liveness as they lead to spurious counterexamples: Message passing FTDAs typically require that a process receives messages from (nearly) all correct processes, which is problematic if processes (i.e., potential senders) or messages are lost.

Besides, in Figure 1 we model message receptions by an update of the variable *rcvd*, more precisely, $\mathit{rcvd} \leq \mathit{rcvd}' \wedge \mathit{rcvd}' \leq \mathit{nsnt} + f$. One may observe that this alone does not require that the value of *rcvd* actually increases. Hence, we add justice requirements, e.g., $J = \{[\forall i.\ \mathit{rcvd}_i \geq \mathit{nsnt}]\}$ in our case study. As observed by [29], counter abstraction may lead to justice suppression. Given a counter-example in the form of a lasso, we detect whether its loop contains only unjust states. If this is the case, similar to an idea from [29], we refine $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by adding a justice requirement, which is consistent with existing requirements in all concrete instances.

Below, we give a general framework for a sound refinement of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$. (In [23], we provide a more detailed discussion on the practical refinement techniques that we use in our experiments.) To simplify presentation, we define a *monster system* as a (possibly infinite) Kripke structure $\mathsf{Sys}_\omega = (S_\omega, S^0_\omega, R_\omega, \mathrm{AP}, \lambda_\omega)$, whose state space and transition relation are disjoint unions of state spaces and transition relations of system instances $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) = (S_{\mathbf{p}}, S^0_{\mathbf{p}}, R_{\mathbf{p}}, \mathrm{AP}, \lambda_{\mathbf{p}})$ over all admissible parameters:

$$S_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S_{\mathbf{p}}, \qquad S^0_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S^0_{\mathbf{p}}, \qquad R_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} R_{\mathbf{p}}$$

$$\lambda_\omega : S_\omega \to 2^{\mathrm{AP}} \text{ and } \forall \mathbf{p} \in \mathbf{P}_{RC}, \forall s \in S_{\mathbf{p}}.\ \lambda_\omega(s) = \lambda_{\mathbf{p}}(s)$$

Let $h\colon S_\omega \to S_{\mathsf{C}}$ be an abstraction mapping, e.g., a combination of the abstraction mappings from Section V.

Definition 12: A sequence $T = \{\sigma_i\}_{i \geq 1}$ is a *concretization* of path $\hat{T} = \{w_i\}_{i \geq 1}$ from $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ if and only if $\sigma_1 \in S^0_\omega$ and for all $i \geq 1$ it holds $h(\sigma_i) = w_i$.

Definition 13: A path $\hat{T}$ of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ is a *spurious path* iff every concretization T of $\hat{T}$ is not a path in Sys_ω.

A prerequisite to abstraction refinement is to check whether a counter-example provided by the model checker is spurious. While for finite state systems there are methods to detect whether a path is spurious [7], we are not aware of a method to detect whether a path $\hat{T}$ in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ corresponds to a path in the (concrete) infinite monster system Sys_ω. Therefore, we limit ourselves to detecting and refining uniformly spurious transitions and unjust states. We first consider spurious transitions.

Definition 14: An abstract transition $(w, w') \in R_{\mathsf{C}}$ is *uniformly spurious* iff there is no transition $(\sigma, \sigma') \in R_\omega$ with $w = h(\sigma)$ and $w' = h(\sigma')$.

The following theorem provides us with a general criterion that ensures that removing uniformly spurious transitions does not affect the property of transition preservation.

Theorem 15: Let $T \subseteq R_{\mathsf{C}}$ be a set of spurious transitions. Then for every transition $(\sigma, \sigma') \in R_\omega$ there is a transition $(h(\sigma), h(\sigma'))$ in $R_{\mathsf{C}} \setminus T$.

It follows that the system $(S_{\mathsf{C}}, S^0_{\mathsf{C}}, R_{\mathsf{C}} \setminus T, \mathrm{AP}, \lambda_{\mathsf{C}})$ still simulates Sys_ω. After considering spurious transitions, we now consider justice suppression.

Definition 16: An abstract state $w \in S_{\mathsf{C}}$ is *unjust under* $q \in \mathrm{AP}_D$ iff there is no concrete state $\sigma \in S_\omega$ with $w = h(\sigma)$ and $q \in \lambda_\omega(\sigma)$.

Consider infinite counterexamples of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$, which have a form of lassos $w_1 \ldots w_k (w_{k+1} \ldots w_m)^\omega$. For such a counterexample $\hat{T}$ we denote the set of states in the lasso's loop by U. We then check, whether all states of U are unjust

under some justice constraint $q \in J$. If this is the case, then $\hat{T}$ is a spurious counterexample, because the justice constraint q is violated. Note that it is sound to only consider infinite paths, where states outside of U appear infinitely often; in fact, this is a justice requirement. To refine C's unjust behavior we add a corresponding justice requirement. Formally, we augment J (and AP_D) with a propositional symbol $[\mathit{off}\ U]$. Further, we augment the labelling function λ_{C} such that every $w \in S_{\mathsf{C}}$ is labelled with $[\mathit{off}\ U]$ if and only if $w \notin U$.

Theorem 17: Let $J \subseteq \mathrm{AP}_D$ be a set of justice requirements, $q \in J$, and $U \subseteq S_{\mathsf{C}}$ be a set of unjust states under q. Let $\pi = \{\sigma_i\}_{i\geq 1}$ be an arbitrary fair path of Sys_ω under J. The path $\hat{\pi} = \{\hat{h}(\sigma_i)\}_{i\geq 1}$ is fair in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ under $J \cup \{[\mathit{off}\ U]\}$.

From this we derive that loops containing only unjust states can be eliminated, and thus $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ be refined.

We encountered cases where several non-uniform spurious transitions resulted in a uniformly spurious path (i.e., a counterexample). We refine such spurious behavior by invariants. These invariants are provided by the user as invariant candidates, and are then automatically checked to actually be invariants using an SMT solver. In our example the invariant is simply "the number of processes that sent a message equals the number of sent messages."

VII. Experimental Evaluation

To show feasibility of our abstractions, we have implemented the PIA abstractions and the refinement loop in OCaml as a prototype tool ByMC. We evaluated it on different broadcasting algorithms. They deal with different fault models and resilience conditions; the algorithms are: (Byz), which is the algorithm from Figure 1, for t Byzantine faults if $n > 3t$, (symm) for t symmetric (identical Byzantine [2]) faults if $n > 2t$, (omit) for t send omission faults if $n > 2t$, and (clean) for t clean crash faults [37] if $n > t$. In addition, we verified the RBC algorithm — formalized also in [18] — whose CFA is given in Figure 2. In this paper we verify the following safety and liveness specifications:

$$[\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC}] \quad \text{(U)}$$

$$[\forall i.\ sv_i = \mathrm{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = \mathrm{AC}] \quad \text{(C)}$$

$$\mathbf{G}\,(\neg\,[\exists i.\ sv_i = \mathrm{AC}]) \vee\ \mathbf{F}\ [\forall j.\ sv_j = \mathrm{AC}] \quad \text{(R)}$$

In addition, in [18] a specification A for RBC was introduced, which we verify for RBC. In contrast to [18], we actually implemented our verification method and give experimental data.

From the literature we know that we cannot expect to verify these FTDAs without restricting the environment, e.g., with communication fairness, namely, every message sent is eventually received. To capture this, we use justice requirements, e.g., $J = \{[\forall i.\ \mathit{rcvd}_i \geq \mathit{nsnt}]\}$ in the Byzantine case.

We extended Promela [22] with constructs to express Π, AP, *RC*, and N [24]. ByMC receives a description of a CFA $\mathcal{A}$ in this extended Promela, and then syntactically extracts the thresholds. The tool chain uses the Yices SMT solver for existential abstraction, and generates the counter abstraction $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ in standard Promela, such that we can use Spin to do finite state model checking. Finally, ByMC also implements the refinements introduced in Section VI and refines the Promela code for $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by introducing predicates capturing spurious transitions and unjust states.

TABLE I. Summary of experiments

$M \models \varphi$?	RC	Spin Time	Spin Memory	Spin States	Spin Depth	$\lvert\widehat{D}\rvert$	#R	Total Time
Byz $\models U$	(A)	2.3 s	82 MB	483k	9154	4	0	4 s
Byz $\models C$	(A)	3.5 s	104 MB	970k	20626	4	10	32 s
Byz $\models R$	(A)	6.3 s	107 MB	1327k	20844	4	10	24 s
Sym $\models U$	(A)	0.1 s	67 MB	19k	897	3	0	1 s
Sym $\models C$	(A)	0.1 s	67 MB	19k	1113	3	2	3 s
Sym $\models R$	(A)	0.3 s	69 MB	87k	2047	3	12	16 s
Omt $\models U$	(A)	0.1 s	66 MB	4k	487	3	0	1 s
Omt $\models C$	(A)	0.1 s	66 MB	7k	747	3	5	6 s
Omt $\models R$	(A)	0.1 s	66 MB	8k	704	3	5	10 s
Cln $\models U$	(A)	0.3 s	67 MB	30k	1371	3	0	2 s
Cln $\models C$	(A)	0.4 s	67 MB	35k	1707	3	4	8 s
Cln $\models R$	(A)	1.1 s	67 MB	51k	2162	3	13	31 s
RBC $\models U$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
RBC $\models A$	—	0.1 s	66 MB	1.7k	333	2	0	1 s
RBC $\models R$	—	0.1 s	66 MB	1.2k	259	2	0	1 s
RBC $\not\models C$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
Byz $\not\models U$	(B)	5.2 s	101 MB	1093k	17685	4	9	56 s
Byz $\not\models C$	(B)	3.7 s	102 MB	980k	19772	4	11	52 s
Byz $\not\models R$	(B)	0.4 s	67 MB	59k	6194	4	10	17 s
Byz $\models U$	(C)	3.4 s	87 MB	655k	10385	4	0	5 s
Byz $\models C$	(C)	3.9 s	101 MB	963k	20651	4	9	32 s
Byz $\not\models R$	(C)	2.1 s	91 MB	797k	14172	4	30	78 s
Sym $\not\models U$	(B)	0.1 s	67 MB	19k	947	3	0	2 s
Sym $\not\models C$	(B)	0.1 s	67 MB	18k	1175	3	2	4 s
Sym $\models R$	(B)	0.2 s	67 MB	42k	1681	3	8	12 s
Omt $\models U$	(D)	0.1 s	66 MB	5k	487	3	0	1 s
Omt $\not\models C$	(D)	0.1 s	66 MB	5k	487	3	0	2 s
Omt $\not\models R$	(D)	0.1 s	66 MB	0.1k	401	3	0	2 s

Table I summarizes our experiments run on 3.3GHz Intel® Core™ 4GB. In the cases (A) we used resilience conditions as provided by the literature, and verified the specification. The model RBC is the reliable broadcast algorithm also considered in [18] under the resilience condition $n \geq t \geq f$. In the bottom part of Table I we used different resilience conditions under which we expected the algorithms to fail. The cases (B) capture the case where more faults occur than expected by the algorithm designer ($f \leq t+1$ instead of $f \leq t$), while the cases (C) and (D) capture the cases where the algorithms were designed by assuming wrong resilience conditions (e.g., $n \geq 3t$ instead of $n > 3t$ in the Byzantine case). We omit (clean) as the only sensible case $n = t = f$ (all processes are faulty) results into a trivial abstract domain of one interval $[0, \infty)$. The column "#R" gives the numbers of refinement steps. In the cases where it is greater than zero, refinement was necessary, and "Spin Time" refers to the Spin running time after the last refinement step. Finally, column $|\widehat{D}|$ indicates the size of the abstract domain.

VIII. Related work

Traditionally, correctness of FTDAs is shown by handwritten proofs [27], [2], and, in some cases, by proof assistants [26], [31], [5]. Completely automated model checking or synthesis are usually not parameterized [24], [36], [34], [3]. Our work stands in the tradition of parameterized model checking for protocols [4], [20], [17], [29], [8], e.g., mutual exclusion and cache coherence. In particular, counter abstraction and justice preservation by Pnueli et al. [29] are keystones of our work.

To the best of our knowledge there are two papers on parameterized model checking of FTDAs [18], [1]. The authors of [18] use regular model checking to make interesting theoretical progress, but did not do any implementation. Their models are limited to processes whose local state space and transition relation are *finite and independent of parameters*. This was sufficient to formalize a reliable broadcast algorithm that tolerates crash faults, and where every process stores whether it has received at least one message. Such models are *not sufficient* to capture FTDAs that contain threshold guards as in our case. Moreover, the presence of a resilience condition such as $n > 3t$ would require them to intersect the regular languages, which describe sets of states, with context-free languages that enforce the resilience condition.

In [1], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. These FTDAs have similar restrictions as the ones considered in [18]: Alberti et al. [1] mention that they did not consider FTDAs that feature "substantial arithmetic reasoning", i.e., threshold guards and resilience conditions, as they would require novel suitable techniques. Our abstractions address this arithmetic reasoning.

To the best of our knowledge, the current paper is thus the first in which *safety* and *liveness* of an FTDA that tolerates Byzantine faults has been *automatically* verified for *all* system sizes and *all* admissible numbers of faulty processes.

IX. Conclusions

We extended the standard setting of parameterized model checking to processes that use threshold guards, and are parameterized with a resilience condition. As a case study we have chosen the core of several broadcasting algorithms under different failure models, including one [33] that tolerates Byzantine faults. These algorithms are widely applied in the literature: typically, multiple (possibly an unbounded number of) instances are used in combination. As future work, we plan to use compositional model checking techniques [28] for parameterized verification of such algorithms. Another open issue is to capture additional fault assumptions such as communication faults [5], [37].

References

[1] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi, "Universal guards, relativization of quantifiers, and failure models in model checking modulo theories," *JSAT*, vol. 8, no. 1/2, pp. 29–61, 2012.

[2] H. Attiya and J. Welch, *Distributed Computing*, 2nd ed. Wiley, 2004.

[3] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant distributed programs," *Distributed Computing*, vol. 25, no. 1, pp. 83–108, 2012.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Inf. Comput.*, vol. 81, pp. 13–31, 1989.

[5] B. Charron-Bost and S. Merz, "Formal verification of a consensus algorithm in the heard-of model," *IJSI*, vol. 3, no. 2–3, pp. 273–303, 2009.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[7] E. 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.

[8] E. Clarke, M. Talupur, and H. Veith, "Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems," in *TACAS'08/ETAPS'08*. Springer, 2008, pp. 33–47.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM TOPLAS*, vol. 16, no. 5, pp. 1512–1542, 1994.

[10] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[12] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM TOPLAS*, vol. 19, no. 2, pp. 253–291, 1997.

[13] R. De Prisco, D. Malkhi, and M. K. Reiter, "On k-set consensus problems in asynchronous systems," *TPDS*, vol. 12, no. 1, pp. 7–21, 2001.

[14] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *J. ACM*, vol. 33, no. 3, pp. 499–516, 1986.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J.ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[16] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *CADE*, ser. LNCS, 2000, vol. 1831, pp. 236–254.

[17] ——, "Exact and efficient verification of parameterized cache coherence protocols," in *CHARME*, ser. LNCS, vol. 2860, 2003, pp. 247–262.

[18] D. Fisman, O. Kupferman, and Y. Lustig, "On verifying fault tolerance of distributed protocols," in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 315–331.

[19] M. Függer and U. Schmid, "Reconciling fault-tolerant distributed computing and systems-on-chip," *Dist. Comp.*, vol. 24, no. 6, pp. 323–355, 2012.

[20] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, pp. 675–735, 1992.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[22] G. Holzmann, *The SPIN Model Checker*. Addison-Wesley, 2003.

[23] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Counter attack on Byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms," *arXiv CoRR*, vol. abs/1210.3846, 2012.

[24] ——, "Towards modeling and model checking fault-tolerant distributed algorithms," in *SPIN*, ser. LNCS, vol. 7976, 2013, pp. 209–226.

[25] Y. Kesten and A. Pnueli, "Control and data abstraction: the cornerstones of practical formal verification," *STTT*, vol. 2, pp. 328–342, 2000.

[26] P. Lincoln and J. Rushby, "A formally verified algorithm for interactive consistency under a hybrid fault model," in *FTCS*, 1993, pp. 402–411.

[27] N. Lynch, *Distributed Algorithms*. Morgan Kaufman, 1996.

[28] K. L. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *CHARME*, ser. LNCS, vol. 2144, 2001, pp. 179–195.

[29] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0,1,∞)- counter abstraction," in *CAV*, ser. LNCS. Springer, 2002, vol. 2404, pp. 93–111.

[30] S. Sankaranarayanan, F. Ivancic, and A. Gupta, "Program analysis using symbolic ranges," in *SAS*, ser. LNCS, vol. 4634, 2007, pp. 366–383.

[31] U. Schmid, B. Weiss, and J. Rushby, "Formally verified Byzantine agreement in presence of link faults," in *ICDCS*, 2002, pp. 608–616.

[32] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, 1987.

[33] T. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Dist. Comp.*, vol. 2, pp. 80–94, 1987.

[34] W. Steiner, J. M. Rushby, M. Sorea, and H. Pfeifer, "Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation," in *DSN*, 2004, pp. 189–198.

[35] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *FMCAD*, 2008, pp. 1–8.

[36] T. Tsuchiya and A. Schiper, "Verification of consensus algorithms using satisfiability solving," *Dist. Comp.*, vol. 23, no. 5–6, pp. 341–358, 2011.

[37] J. Widder and U. Schmid, "Booting clock synchronization in partially synchronous systems with hybrid process and link failures," *Dist. Comp.*, vol. 20, no. 2, pp. 115–140, 2007.

Verifying Multi-threaded Software with Impact

Björn Wachter
Department of Computer Science
University of Oxford
Email: bjoern.wachter@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science
University of Oxford
Email: daniel.kroening@cs.ox.ac.uk

Joël Ouaknine
Department of Computer Science
University of Oxford
Email: joel.ouaknine@cs.ox.ac.uk

***Abstract*—Lazy abstraction with interpolants, also known as the *Impact algorithm*, is en vogue as a state-of-the-art software model-checking technique for sequential programs. However, a direct extension of the Impact algorithm to concurrent programs is bound to be inefficient as it has to explore all thread interleavings, which leads to control-state explosion. To this end, we present a new algorithm that combines a new, symbolic form of partial-order reduction with Impact. Our algorithm carries out the dependence analysis on-the-fly while constructing the abstraction and is thus able to deal precisely with dynamic dependencies arising from accesses to tables or pointers — a setting where classical static partial-order reduction techniques struggle. We have implemented the algorithm in a prototype tool that analyses concurrent C program with POSIX threads and evaluated it on a number of benchmark programs. To our knowledge, this is the first application of an Impact-like algorithm to concurrent programs.**

I. Introduction

Concurrent software is gaining importance owing to the advent of power-efficient multi-core architectures. Model checking for concurrent software is thus one of the most pressing problems facing the verification community. Concurrent software in C/C++ is usually written using mainstream APIs such as POSIX, or via a combination of language and library support as in Java. Typically, multiple threads are spawned—either up-front or dynamically—which communicate via shared variables. While software verification generally has to cope with *data state explosion*, threads introduce the problem of state explosion due to the need of keeping track of a plethora of thread interleavings.

Lazy abstraction with interpolants [1], also known as the *Impact algorithm*, has emerged as one of the most efficient algorithms for addressing the data state explosion problem for sequential programs. *Impact* unwinds the control-flow graph of the program in the form of an abstract reachability tree. Whenever the exploration arrives at an error state, the nodes on the error path are annotated with invariants that prove infeasibility of the error path. The crux of the algorithm is a covering check that allows the algorithm to soundly stop the unwinding and terminate with a correctness proof of the program. The underlying observation is that tree nodes represent sets of program states which are related by subset relations. Roughly, a node w labeled with $x > 0$ "contains" a node v labeled with $x > 1$. If we have established that the superset node w cannot be on an error path, we do not need to search for an error path from subset node v. This combination of low-cost program unwindings combined with path-based refinement and covering checks gives rise to an efficient software model checking algorithm.

Supported by ERC project 280053, EPSRC project EP/H017585/1 and the Semiconductor Research Corporation (SRC) under task 2269.002.

main()	thread T_1	thread T_2

`assume(i!=j);`
`v[i]=0; v[j]=0;`
`pthread_create(`T_1`);`
`pthread_create(`T_2`);`
`pthread_join(`T_1`);`
`pthread_join(`T_2`);`
`assert(v[j] ≥ 0);`
	A: `v[i]=1;`
	B: `v[i]=v[i]+1;`
	C: `v[i]=v[j];`
		a: `v[j]=-2;`
		b: `v[j]=v[j]+1;`
		c: `v[i]=v[i]+1;`

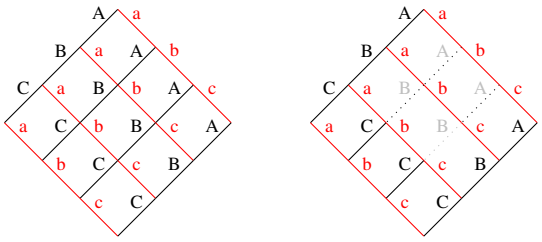


Fig. 1: An example program (top) and its complete interleaving (left) and reduced interleaving semantics (right).

However, the original *Impact* algorithm has been devised for sequential code only. A direct extension of *Impact* to multi-threaded programs amounts to an enumeration of thread interleavings. Let us illustrate this with the example program with two threads given in Figure 1. On the left-hand side of the figure, the state graph with the complete set of interleavings is shown. Note that there is a diamond-shaped structure where program paths merge, e.g., executing instruction A and then a leads to the same state as executing a first and then A, making certain sequences of instructions redundant. This situation is very common in multi-threaded programs.

Impact produces the full program unwinding, as the exploration of the abstract tree has to reach an error location to discover the right invariants. The algorithm may find identical invariants for redundant paths, but this does not prune the abstract exploration, as, at that point, the program paths have already been completely unwound.

Force cover, an optimization of *Impact*, improves this situation by giving *Impact* the power to discover that certain program executions merge without fully exploring the paths to the error location. This reduces the number of paths to be explored. On our example, the application of force covers results in a tree of a similar size as the graph on the left-hand

side of Figure 1. In particular, even with force covers, *Impact* still explores *all* thread interleavings in our example, which can be prohibitively expensive.

A principal method to reduce the number of interleavings in the exploration of concurrent programs is *partial-order reduction* (POR) [2]–[5]. The right-hand side of Figure 1 shows an exploration reduced by means of partial-order reduction. A key contribution of this paper is a novel combination of *Impact* with POR, which produces the abstract tree shown in Figure 2. *Impact* with force cover alone explores a tree with five further nodes, as it does not know in advance that the executions merge, while partial-order reduction is able to discover this earlier. Discovering redundant paths early on during the exploration is crucial to avoid path explosion.

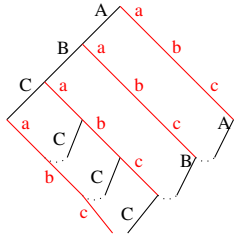


Fig. 2: *Impact* with POR and force cover

Contributions:

- We present an extension of the Impact algorithm to concurrent software.
- We show how to combine partial-order reduction with Impact. Due to a subtle interplay between node coverings and POR, obtaining a sound verification procedure is non-trivial. To this end, we give a general framework to prove such combinations correct, and an algorithm based on this framework which combines Impact with monotonic partial-order reduction [6].
- We compare the effect of partial-order reduction and force covers; our conjecture is that the two techniques yield orthogonal benefits and are best combined.

We present background and basic definitions in Section II. We develop a variant of the *Impact* algorithm for concurrent software in Section III. We present a combination of partial-order reduction and *Impact* in Section IV. Experimental results are discussed in Section V.

II. Basic Definitions

Program semantics: We consider a concurrent program $\mathscr{P}$ composed of a finite set of threads $\mathscr{T}$, which communicate by performing operations on shared variables.

A state of a concurrent system consists of the local states S_{local} of each thread, i.e., the value of the thread's program counter given by a program location $l \in L$ and values of the local variables of the thread, and of the shared states S_{shared}, i.e., values for communication objects such as locks, tables and the like. Thus, we have a global state space $S = S_{shared} \times S_{local}$.

A global control location is a function $\mathfrak{l} : \mathscr{T} \to L$ from threads to control locations. Let L_G be the set of global control locations. The global location in state s is denoted by $\mathfrak{l}(s)$. For a given global location $\mathfrak{l}$ and thread T, we write $\mathfrak{l}_T$ as a shorthand for $\mathfrak{l}(T)$. By $\mathfrak{l}[T \mapsto l]$, we denote the global location where the location of thread T maps to l, while the locations for all other threads T' remain unchanged.

We characterize program data in terms of formulas in standard first-order logic. We denote the set of well-formed formulas over symbols Σ by $\mathscr{F}(\Sigma)$. For a given formula F we denote the set of formulas over the same symbols by $\mathscr{F}(F)$.

Let V be the vocabulary that represents the program variables. A state formula is a formula in $\mathscr{F}(V)$ and represents a set of global states. A transition formula, from now on, typically denoted by the letter R, is a formula in $\mathscr{F}(V \cup V')$.

Formally, we model a program as a pair $(init, \mathscr{T})$ where $init \in \mathscr{F}(V)$ is the initial-state predicate, and $\mathscr{T}$ a finite set of threads. We assume that the set of threads is endowed with some total order $<$. A thread $T \in \mathscr{T}$ is a tuple $T = (L, l^i, l^\lightning, A)$ consisting of a finite set of control locations L, an initial location $l^i \in L$, an error location $l^\lightning$, and a set of actions A. An action is a pair $a = (l, N) \in L \times 2^{\mathscr{F}(V \cup V') \times L}$, consisting of a current location l and a set of successor control locations l', each associated with a transition constraint. An assignment l_1: `x=y+1;` l_2: … is represented as $(l_1, \{(x' = y + 1 \wedge y' = y; l_2)\})$. An assertion l_1: `assert(x<y);` l_2: … becomes an action $(l_1, \{(x \geq y \wedge x' = x \wedge y' = y, l^\lightning), (x < y \wedge x' = x \wedge y' = y, l_2)\})$, which enters the error location $l^\lightning$ if the condition is violated. Sets of successors are used to represent branching control flow, e.g., the encoding of the `if`-statement l_1: `if(x==1) goto` l_3; l_2: … is $(l_1, \{(x = 1 \wedge x' = x, l_3), (x \neq 1 \wedge x' = x, l_2)\})$.

We write $L(T)$ and $A(T)$ to denote the locations and actions of a thread. For an action $a = (l, N) \in A(T)$ of thread T, action a is enabled at location l and at global location $\mathfrak{l}$ if a is enabled at $\mathfrak{l}_T$. We assume that exactly one action $a_{T,l}$ of any given T is enabled at any location $l \in L$.

The control-flow graph $CFG_T = (l^i, E)$ of thread $T = (L, l^i, l^\lightning, A)$ is defined by entry node l^i and edges $E = \bigcup_{a \in A(T)} E_a$ where $E_a = \{(l, l') \in L \times L \mid a = (l, N), (R, l') \in N\}$. The control-flow nodes are topologically ordered. We say that an action a induces a back edge if E_a contains a back edge.

We say that an action $a = (l, N) \in T$ is enabled at a state if a is enabled at global location $\mathfrak{l}(s)$. We denote the enabled actions at a state s by $enabled(s)$. We assume that an action $a = (l, N)$ defines a total function $\{s \in S \mid a \in enabled(s)\} \to S$ on all program states for which it is enabled.

For ease of notation, we identify a with this function and write $a(s)$ to denote the successor of a state s under action a.

Invariants and Correctness Proofs: A program path π is a sequence $(\mathfrak{l}_0, T_0, a_0, \mathfrak{l}_1) \ldots (\mathfrak{l}_{N-1}, T_{N-1}, a_{N-1}, \mathfrak{l}_N)$. For a thread T, and $l, l' \in L(T)$ with $l \neq l'$, we write $l \sqsubset l'$ if there exists a program path from l to l'.

A path is an *error path* if $\mathfrak{l}_0$ is the vector of initial locations for all threads, and $\mathfrak{l}_{N-1}$ contains an error location of a thread. We denote by $\mathscr{F}(\pi)$ the sequence of formulas $init^{(0)} \wedge R_0^{(0)}, \ldots R_{N-1}^{(N-1)}$ obtained by shifting each R_i i time frames into the future. We say that π is feasible if $\bigwedge R_i^{(i)}$ is logically satisfiable. A solution to $\bigwedge R_i^{(i)}$ corresponds to a program execution assigning values to the program variables at each execution step. The program is said to be safe if all error paths are infeasible.

An *inductive invariant* is a mapping $I : L_G \to \mathscr{F}(V)$ such that $init \Rightarrow I(\mathfrak{l}^i)$ and for all locations $\mathfrak{l} \in L_G$, all threads $T \in \mathscr{T}$,

and actions $a = (l, R, l') \in T$ in thread T enabled in $\mathfrak{l}$, we have $I(\mathfrak{l}) \wedge R \Rightarrow I(\mathfrak{l}[T \mapsto l'])$. A *safety invariant* is an inductive invariant with $I(\mathfrak{l}) \equiv \mathit{False}$ for all error locations $\mathfrak{l}$. If there is a safety invariant the program is safe.

Interpolants: In case a path is infeasible, an explanation can be extracted in the form of an interpolant. To this end, we define *sequent interpolants* [7]. A sequent interpolant for formulas $A_1, \ldots, A_N$, is a sequence $\widehat{A}_1, \ldots, \widehat{A}_N$ where the first formula is equivalent to true $\widehat{A}_1 \equiv \mathit{True}$, the last formula is equivalent to false $\widehat{A}_N \equiv \mathit{False}$, consecutive formulas imply each other, i.e., for all $i \in \{1, \ldots, N\}$, $\widehat{A}_{i-1} \wedge A_i \Rightarrow \widehat{A}_i$, and, the i-th sequent is a formula over the common symbols of its prefix and postfix, i.e., for all $i \in \{1, \ldots N\}$, $\widehat{A}_i \in \mathscr{F}(A_1, \ldots, A_i) \cap \mathscr{F}(A_{i+1}, \ldots, A_N)$. For certain theories, quantifier-free interpolants can be generated for inconsistent, quantifier-free sequences $A_1, \ldots, A_N$ [7].

III. Impact Algorithm for Concurrent Programs

We now present an extension of the original Impact algorithm to concurrent programs. The algorithm returns either a safety invariant for a given program, finds a counterexample or diverges (the verification problem is undecidable). To this end, the algorithm constructs an abstraction of the program in the form of an abstract reachability tree, which corresponds to a program unwinding annotated with invariants.

Definition 3.1 (ART): An *abstract reachability tree* (ART) $\mathscr{A}$ for program $\mathscr{P}$ is a tuple $(V, \epsilon, \rightarrow, \triangleright)$ consisting of a tree with nodes V, root node $\epsilon \in V$, edges $\rightarrow \subseteq V^2$, and a covering relation $\triangleright \subseteq V^2$ between tree nodes such that:

- every nodes $v \in V$ is labeled with a tuple $(\mathfrak{l}, \phi)$ consisting of a current global control location $\mathfrak{l}$, and a state formula ϕ. We write $\mathfrak{l}(v)$ and $\phi(v)$ to denote the control location and annotation, respectively, of node v.
- edges correspond to program actions, and tree branching represents both branching in the control flow within a thread and thread interleaving. Formally, an edge is a tuple (v, T, R, w) where $v, w \in V$, $T \in \mathscr{T}$, and R the transition constraint of the corresponding action.

We write $v \xrightarrow{T} w$ if there exists an edge $(v, T, R, w) \in \rightarrow$. We denote by $\rightsquigarrow$ the transitive closure of $\rightarrow$.

To put abstract reachability trees to work for proving program correctness for unbounded executions, we need a criterion to prune the tree without missing any error paths. This role is assumed by the covering relation $\triangleright$.

Intuitively, the purpose of node labels is to represent inductive invariants, i.e., over-approximations of sets of states, and the covering relation is the equivalent of a subset relation between nodes. Suppose that two nodes v, w share the same control location, and $\phi(v)$ implies $\phi(w)$. If there was a feasible error path from v, there would be a feasible error path from w. Therefore, if we can find a safety invariant for w, we do not need to explore successors of v, as $\phi(v)$ is at least as strong as the already sufficient invariant $\phi(w)$.

Note that, therefore, if w is safe, all nodes in the subtree rooted in v are safe as well. Therefore, a node is covered if and only if the node itself or any of its ancestors has a label implied by another node's label at the same control location.

To obtain a proof from an ART, the ART needs to fulfill certain conditions, summarized in the following definition:

Definition 3.2 (Safe ART): Let $\mathscr{A} = (V, \epsilon, \rightarrow, \triangleright)$ be an ART.

- $\mathscr{A}$ is *well-labeled* if the labeling is inductive, i.e., $\forall (v, T, R, w) \in \rightarrow\colon\ \mathfrak{l}(v) = \mathfrak{l}(w) \wedge \phi(v) \wedge R \Rightarrow \phi(w)'$ and compatible with covering, i.e., $(v, w) \in \triangleright\colon\ \phi(v) \Rightarrow \phi(w)$ and w not covered.
- $\mathscr{A}$ is *complete* if all of its nodes are covered, or have an out-going edge for every action that is enabled at $\mathfrak{l}$.
- $\mathscr{A}$ is *safe* if all error nodes are labeled with *False*.

Theorem 3.3: If there is a safe, complete, well-labeled ART of program $\mathscr{P}$, the program is safe.

Proof As in [1], the labeling immediately gives a safety invariant M, $M(\mathfrak{l}') = \bigvee \{\phi(v) \mid \mathfrak{l}(v) = \mathfrak{l}'\}$. ∎

A. Concurrent Impact with Full Interleaving

The concurrent version of the Impact algorithm we describe next (Algorithm 1) constructs an ART by alternating three different operation on nodes: Expand, Refine, and Close. At all times, the algorithm maintains the invariant that the tree is well-labeled and safe, i.e., to produce a correctness proof the algorithm needs to make the tree complete.

To keep track of nodes where the tree is incomplete, uncovered leaf nodes are kept in a work list Q.

Expand takes an uncovered leaf node and computes its successors. To this end, it iterates over all threads. For every enabled action, it creates a fresh tree node w, and sets its location to the control successor l' given by the action. To ensure that the labeling is inductive, the formula $\phi(w)$ is set to *True*. Then the new node is added to the work list Q. Finally, a tree edge is added (Line 23), which records the step from v to w and the transition formula R. Note that if w is an error location, the labeling is not safe; in which case, we need to refine the labeling, invoking operation Refine.

Refine takes an error node v and, detects if the error path is feasible and, if not, restores a safe tree labeling. First, it determines if the unique path π from the initial node to v is feasible by checking satisfiability of $\mathscr{F}(\pi)$. If $\mathscr{F}(\pi)$ is satisfiable, the solution gives a counterexample in the form of a concrete error trace, showing that the program is unsafe. Otherwise, an interpolant is obtained, which is used to refine the labeling. Note that strengthening the labeling may destroy the well-labeledness of the ART. To recover it, pairs $w \triangleright v_i$ for strengthened nodes v_i are deleted from the relation, and the node w is put into the work list again.

Close takes a node v and checks if v can be added to the covering relation. As potential candidates for pairs $v \triangleright w$, it only considers nodes created before v, denoted by the set $V^{\prec v} \subsetneq V$. This is to ensure stable behavior, as covering in arbitrary order may uncover other nodes, which may not terminate. Thus only for uncovered nodes $w \in V^{\prec v}$, it is checked if $\mathfrak{l}(w) = \mathfrak{l}(v)$ and $\phi(v)$ implies $\phi(w)$. If so, (v, w) is added to the covering

Algorithm 1 Impact with support for concurrent programs

```
 1: procedure MAIN()
 2:     Q := {ε}, ▷ := ∅
 3:     while Q ≠ ∅ do
 4:         select and remove v from Q
 5:         CLOSE(v)
 6:         if v not covered then
 7:             if error(v) then
 8:                 REFINE(v)
 9:             EXPAND(v)
10:     return 𝒫 is safe
11:
12: procedure EXPAND(v)
13:     for T ∈ 𝒯 do
14:         EXPAND-THREAD(T, v)
15: procedure EXPAND-THREAD(T, v)
16:     (l, φ) := v
17:     for (l, N) ∈ A(T) with l_T = l do
18:         for (R, l') ∈ N do
19:             w := fresh node
20:             l(w) := l[T ↦ l']
21:             φ(w) := True
22:             Q := Q ∪ {w}, V := V ∪ {w}
23:             →:=→ ∪{(v, T, R, w)}
24:
25: procedure CLOSE(v)
26:     for w ∈ V^{<v} : w uncovered do
27:         if l(v) = l(w) ∧ φ(v) ⇒ φ(w) then
28:             ▷ := ▷ ∪ {(v, w)}
29:             ▷ := ▷ \ {(x, y) ∈ ▷ | v ⇝ y}
30: procedure REFINE(v)
31:     if v not error node or φ(v) ≡ False then
32:         return
33:     π := v_0, ... v_N path from ε to v
34:     if 𝒮(π) has interpolant A_0 ... A_N then
35:         for i = 0...N do
36:             φ := A_i^{-i}
37:             if φ(v_i) ⊭ φ then
38:                 Q := Q ∪ {w | w ▷ v_i}
39:                 ▷ := ▷ \ {(w, v_i) | w ▷ v_i}
40:                 φ(v_i) := φ(v_i) ∧ φ
41:         for w ∈ V s.t. w ⇝ v do
42:             CLOSE(w)
43:     else
44:         abort (program unsafe)
```

relation $\triangleright$. To restore well-labeling, all pairs (x, y) where y is a descendant of v, denoted by $v \rightsquigarrow y$, are removed from $\triangleright$, as v and all its descendants are covered.

MAIN first initializes the queue with the initial node ϵ, and the relation $\triangleright$ with the empty set. It then runs the main loop of the algorithm until Q is empty, i.e., until the ART is complete, unless an error is found which exits the loop. In the main loop, a node is selected from Q. First, CLOSE is called to try and cover it. If the node is not covered and it is an error node, REFINE is called. Finally, the node is expanded, unless it was covered, and evicted from the work list.

An important optimization of the algorithm is another subroutine, called *force cover*. Initially, all new nodes are labeled with invariant *True*. Therefore, they will not be covered by an existing node with a non-trivial invariant, although this may be a permissible labeling. To check coverage, force cover finds the nearest common ancestor of two nodes and then checks the characteristic formula to the new node to see if the invariant of the other node also holds at the new node. Beyer [8] showed that this optimization is essential for the performance of *Impact*.

Wrapping up the extension of the original Impact algorithm to concurrent programs: the single control location becomes a vector, and the EXPAND routine enumerates all possible interleavings. This algorithm is very inefficient in its basic form: due to the full interleaving semantics, the number of global control locations grows very quickly. We shall amend this in the next section.

IV. PARTIAL ORDER REDUCTION

Performing a thread interleaving at every step would be prohibitively expensive. *Impact* needs some way of reducing interleaving. Therefore, we present an algorithm that combines partial-order reduction with the *Impact* algorithm. A very simple kind of partial order reduction is to only allow interleaving when shared-variable accesses occur, however a much stronger reduction is possible in many cases. In this section, we consider a more advanced partial exploration strategy that generates monotonic program paths Π_{mono}, wherein consecutive independent actions only occur in the order of increasing thread ids [6].

Recall that the soundness proof of the original IMPACT algorithm rests on three pillars, namely: completeness, safety and well-labeledness of ARTs. However, partial order reduction clashes with the original completeness criterion of IMPACT that requires the very thing we aim to avoid: full expansion of all thread interleavings. Thus we need a new soundness proof and, in particular, a weaker completeness criterion, to combine abstraction with partial-order reduction.

To this end, we introduce the new concept of Π-*completeness*, which is parameterized with an exploration strategy via a set of program paths Π, and gives a systematic framework to combine abstraction with partial-order reduction. Based on this concept, we also present the dPOR-IMPACT algorithm, which explores monotonic paths and produces Π_{mono}-complete ARTs.

Before we come to Π-completeness and dPOR-IMPACT, we first need to review some basic POR concepts and notation.

A. Independence and Mazurkiewicz Equivalence

Partial-order reduction is based on the notion of independence of actions. Intuitively, two actions are independent if they commute and we can execute them in any order:

Definition 4.1 (Independence): Two actions a_1 and a_2 are *independent*, denoted by $a_1 \parallel a_2$, if for all states $s \in S$ where a_1 and a_2 are co-enabled, i.e., $a_1, a_2 \in enabled(l(s))$, we have $a_1(a_2(s)) = a_2(a_1(s))$. Otherwise, we say that they are dependent and write $a_1 \not\parallel a_2$.

Partial-order reduction techniques are based on finding a representative subset of the interleavings avoiding the exploration of all equivalent interleavings, i.e., interleavings that lead to equivalent orderings of actions. This leads to the notion of Mazurkiewicz equivalence [9]:

Definition 4.2 (Mazurkiewicz equivalence): Two program paths are *Mazurkiewicz equivalent* if they result from exchanging the order of two independent actions.
We call a set of program paths Π *representative* if it contains a representative path for every Mazurkiewicz equivalence class.

An example for a representative set of program paths are the monotonic program paths, which are defined as follows:

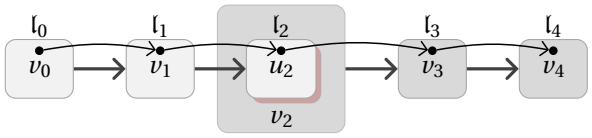


Fig. 3: Path correspondence. Rounded rectangles represent ART nodes $v_0, \ldots, v_4$ and u_2. We have $u_2 \rhd v_2$. The gray arrows depict ART edges. The path $l_0 \ldots l_5$ is a control flow path.

Definition 4.3 (Monotonic paths): A program path $\pi = (\ell_0, T_0, a_0, \ell_1) \ldots (\ell_{N-1}, T_{N-1}, a_{N-1}, \ell_N)$ is *monotonic* if for all $i, j \in \{0, \ldots, N-1\}$ with $i < j$, $a_i \parallel a_j$ and $T_i > T_j$, we have $j \neq i+1$. Let Π_{mono} be the set of monotonic program paths.

B. Π-*completeness*

We will say that an ART $\mathscr{A}$ is Π-*complete* with respect to a set of program paths Π if each path $\pi \in \Pi$ is covered by $\mathscr{A}$. Intuitively, a program path is covered if there exists a corresponding sequence of nodes in the tree, where corresponding means that it visits the same control locations and takes the same actions. In absence of covers, the matching between control paths and sequences of nodes is straightforward.

However, a path of the ART may end in a covered node. For example, consider the path $\ell_0 \ldots \ell_5$ in Figure 3. While prefix $\ell_0\ell_1\ell_2$ can be matched by node sequence $v_0 v_1 u_2$, node u_2 is covered by node v_2, formally $u_2 \rhd v_2$. But how we can match the remainder of the path? We are stuck at node u_2, a leaf with no out-going edges. Our solution is to allow the corresponding sequence to "climb up" the covering order $\rhd$ to a more abstract node, here we climb from u_2 to v_2. Node v_2 in turn must have a corresponding out-going edge, as it cannot be covered and its control location is also ℓ_2. Finally, the corresponding node sequence for $\ell_0 \ldots \ell_4$ is $v_0 \ldots v_4$.

Figure 4 illustrates the formalization of our notion of path correspondence. On top of the figure, we depict a fragment of a program path with locations l_i, l_{i+1} and l_{i+2}, and, at the bottom, the corresponding path which climbs from node u_{i+1} to node v_{i+1} where u_{i+1} and v_{i+1} are both at location $\ell(u_{i+1}) = \ell(v_{i+1}) = \ell_{i+1}$ and $u_{i+1} \rhd v_{i+1}$. A corresponding path is allowed to climb up not only at one position i but at any position i (or none) and at arbitrarily many positions.

This notion is formalized in the following definition:

Definition 4.4 (Corresponding paths & path cover): Consider a program $\mathscr{P}$. Let $\mathscr{A}$ be an ART for $\mathscr{P}$ and let $\pi = (\ell_0, T_0, a_0, \ell_1) \ldots (\ell_{N-1}, T_{N-1}, a_{N-1}, \ell_N)$ be a program path. A *corresponding path* for π in $\mathscr{A}$ is a sequence $v_0, \ldots, v_n$ in $\mathscr{A}$ such that, for all $i \in \{0, \ldots, N-1\}$, $\ell(v_i) = \ell_i$, and

$$\exists u_{i+1} \in V : \; v_i \overset{T_i, a_i}{\rightarrow} u_{i+1} \wedge (u_{i+1} = v_{i+1} \vee u_{i+1} \rhd v_{i+1})$$

A program path π is covered by $\mathscr{A}$ if there exists a corresponding path $v_0, \ldots, v_n$ in $\mathscr{A}$.

We are now ready to define our new completeness criterion:

Definition 4.5 (Π-completeness): Let $\mathscr{P}$ be a program and Π a set of program paths. ART $\mathscr{A}$ for $\mathscr{P}$ is Π-complete if every path $\pi \in \Pi$ is covered by $\mathscr{A}$.

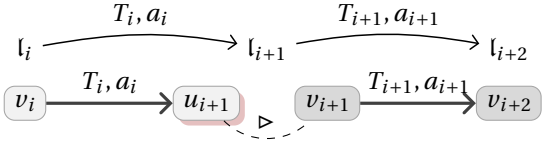


Fig. 4: Illustration of Definition 4.4. The diagram shows a fragment of an ART $\mathscr{A}$ with notation for the nodes of the definition. The dashed line represents a covering edge.

A Π-complete, safe, well-labeled ART constitutes a proof of program correctness, as stated in the following proposition:

Proposition 4.6: Let $\mathscr{P}$ be a program. Let Π be a representative set of program paths. Assume that $\mathscr{A}$ is safe, well-labeled and Π-complete. Then program $\mathscr{P}$ is safe.

C. Abstraction Algorithm

We now combine POR with IMPACT. The obvious starting point is to modify the EXPAND function in Algorithm 1. We first introduce the modified expansion function EXPAND$_\lozenge$. However, changing only the expansion function turns out to be insufficient. Due to a subtle interplay between coverings and POR, the resulting algorithm does not guarantee Π_{mono}-completeness, and is unsound, which we illustrate with a small example. We then describe a method to fix this problem and present Algorithm 2, a sound variant of Impact with POR.

First, we change EXPAND such that only monotonic paths are unwound. To this end, instead of expanding all threads at a node, EXPAND$_\lozenge$ first checks if expanding with T yields a non-monotonic program path. This check is carried out in function SKIP$_\lozenge$ for given node v and thread T. Function SKIP$_\lozenge$ analyses the thread T' and action a' executed by the parent u of v, and returns true if the thread T is smaller than $T < T'$ and action a' is independent of a.

Algorithm 2 dPOR-IMPACT

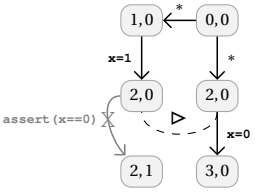
```
1:  procedure EXPAND_◇(v)
2:      for T ∈ 𝒯 with ¬SKIP_◇(v, T) do
3:          EXPAND-THREAD(T, v)
4:
5:  procedure SKIP_◇(v, T)
6:      choose unique T', a' s.t. u -T',a'-> v
7:      return (T < T' ∧ (ACTION(v, T) || a')) ∧ ¬LOOP(u, T')
8:
9:  procedure CLOSE_◇(v)
10:     for w ∈ V^≺v : w uncovered do
11:         if ℓ(v) = ℓ(w) ∧ φ(v) ⇒ φ(w) then
12:             ⊳ := (⊳ ∪ {(v, w)}) \ {(x, y) ∈ ⊳ | v ⇝ y}
13:             for T with v -T,...-> v' and not w -T,...-> w' do
14:                 EXPAND-THREAD(T, w)
```

Intuitively, two writes to the same variable are dependent, a read and a write to the same variable are dependent, but two reads to the same variable are independent. Two actions a and a' are independent, denoted by $a \parallel a'$, if $R_a \cap W_{a'} = \emptyset \wedge W_a \cap (R_{a'} \cup W_{a'}) = \emptyset$ where R_a and $R_{a'}$ are the variables being read, and, W_a and $W_{a'}$ the variables being written.

Additionally, we introduce function LOOP to detect control-flow loops. Function LOOP(u, T) returns true if action $a = (l, N)$

of T at node u induces a back edge in the thread's control flow. This completes our discussion of EXPAND$_\Diamond$.

As mentioned before, just modifying EXPAND yields an unsound algorithm that does not guarantee Π_{mono}-completeness. Consider the example program below. Note that to violate the assertion, the context switch between the two threads has to happen right after T_1 has executed `x=1`. However, the covering between the left and the right $(2,0)$-node prevents this expansion, leading to an ART that is not Π_{mono}-complete. In particular, the counterexample path is not covered by the resulting ART, i.e., there is no corresponding path, as `assert(x==0)` is not expanded at the covering $(2,0)$-node.



T_1	T_2
0: if(*) 1: x=1; 2: x=0; 3:	0: assert(x==0); 1:

the counterexample is
T_1: *, T_1: x=1;, T_2: assert(x==0)

To guarantee Π_{mono}-completeness, we modify CLOSE to carry out expansions at the covering node, so-called cover expansions – yielding function CLOSE$_\Diamond$. We consider actions that would have been expanded at the covered node, had there been no cover. These actions are now expanded in the covering node. In our example, this results in an expansion of `assert(x==0)` on the right $(2,0)$-node, which triggers a refinement that uncovers the left $(2,0)$-node and reveals the counterexample in the next step.

This combination of EXPAND$_\Diamond$ and CLOSE$_\Diamond$ guarantees Π_{mono}-completeness, as proved in the following lemma, which also establishes the correctness of dPOR-Impact:

Lemma 4.7: If Algorithm 2 reports that the program is safe, the computed ART $\mathscr{A}$ is Π_{mono}-complete.

Proof We need to show that every path $\pi \in \Pi_{mono}$ is covered by $\mathscr{A}$. We carry out a proof by induction on the length $N = |\pi|$ of π. The base case for $N = 1$ is trivial. Assume that $N \geq 2$ and that every path of length at most $N-1$ is covered. Let $\pi = (\mathfrak{l}_0, T_0, a_0, \mathfrak{l}_1) \ldots (\mathfrak{l}_{N-1}, T_{N-1}, a_{N-1}, \mathfrak{l}_N) \in \Pi_{pm}$ be a path of length N. We need to prove that there exists a corresponding path $v_0, \ldots, v_N$ that meets the criteria of Definition 4.4.

By induction hypothesis, there exists a corresponding path $v'_0, \ldots, v'_{N-1}$ for the length $N-1$ prefix of π. As $\pi \in \Pi_{mono}$, we have that SKIP$_\Diamond(v'_{N-1}, T_{N-1}) = False$. Hence, if v'_{N-1} is not covered, it will be expanded yielding a suitable successor v'_N, and choosing $v_i = v'_i$ for all $i \in \{1, \ldots, N\}$ we are done. So let us assume that v_{N-1} is covered. Then there exists v_{N-1} distinct from v_{N-1} such that $v_{N-1} \rhd v_{N-1}$ and v_{N-1} is not covered. It could be that SKIP$_\Diamond(v_{N-1}, T_{N-1}) = True$, however the covering $v'_{N-1} \rhd v_{N-1}$ must result from an invocation of CLOSE$_\Diamond$, forcing expansion of T_{N-1} at v_{N-1} and thus yields a suitable successor v_N. Thus we choose $v_i = v'_i$ for $i \in \{1, \ldots, N-2\}$, v_{N-1} and v_N as above, and $u_{N-1} = v'_{N-1}$. ∎

D. Conditional Dependence

We now describe how to deal with aliasing in presence of pointers and shared tables. This leads to dynamic dependencies determined by the execution state, e.g., when dereferencing pointers, the dependence relation is determined by pointer aliasing. Two pointer variables may point to the same location leading to a dependency, or to disjoint locations. When accessing tables via indices, dependencies may arise when two threads access the same position in a table, which depends on the value of the indexing variable.

Dynamic dependencies can be accommodated in our framework by considering so-called *conditional dependence* between actions [2]. Effectively, the dependence relation, which was a binary relation between actions until now, becomes a ternary relation, such that dependencies are triples consisting of a state and two actions. When carrying out partial-order reduction, the ART is built in the same way as before, except that the dependency check takes into account the aliasing information.

Computation of the aliasing information can be carried out by simply inspecting the history of the state. However, note that covering produces nodes that represent states with potentially different histories. Hence, if aliasing information is used to prune expansions, this alias information must also be annotated in the node labels, to ensure soundness. This can be achieved as follows: we carry out a simple aliasing analysis along the history of a node, if we find that there is no aliasing (and hence no dependence), we refine the nodes along the path with inductive invariants that enforce absence of the alias. For a pair of accesses, we define an alias expression $alias$, such that the expression becomes true if and only if the two accesses go to the same address. The construction of alias expressions for typical array accesses is described, e.g., in [6].

For illustration, consider the example in Figure 1. For the path aA, we need to check independence of the access `v[i]=2` and `v[j]=-2`, which gives the alias expression $i = j$. Let π be the path to the node at which we check the alias relation, in our example aA. The accesses are independent if the conjunction of path formula and alias expression $\mathscr{F}(\pi) \wedge alias^{(|\pi|-1)}$ is unsatisfiable. In our example, this formula is unsatisfiable, due to the assume statement in line 1 of `main`, and the nodes along the path are refined with the interpolant $i \neq j$, and we can make the reduction depicted in the figure.

V. EXPERIMENTS

We have implemented the techniques described in this paper in a prototype tool, called IMPARA, a software model checker for concurrent C programs with POSIX or WIN32 threads. Experiments were run on an Intel Xeon machine with 8 cores at 3.07 GHz with 50 GB RAM. The timeout is 900 s and the memory limit is 15 GB. We make the implementation and detailed results available online at http://www.cprover.org/concurrent-impact/ for evaluation.

Comparison with Other Tools: We compare the performance of IMPARA 0.2 with the tools CBMC 4.5 [10] (bounded model checking with partial-order encoding), ESBMC 1.20 [11] (bounded model checking, POR and state

program	safe	CBMC	ESBMC	SATABS	THREADER	IMPARA
dekker	y	0.6*	2.2*	0.2	TO	**0.1**
lamport	y	12.4*	18.1*	**0.3**	38.1	**0.3**
peterson	y	0.2*	2.0*	0.3	4.8	**0.1**
szymanski	y	0.5*	4.7*	**0.2**	13.5	**0.2**
read_write_u	n	**0.2**	TO	0.8	58.4	0.6
read_write_s	y	**0.4**	TO	0.8	58.1	0.9
time_var_mutex	y	0.2	110.3	95.4	4.3	**0.1**
stack_u	n	1.0	TO	TO	80.6	**0.5**
stack_s	y	**33.5**	TO	TO	250.1	38.8

TABLE I: IMPARA vs. other tools on competition benchmarks

hashing), THREADER 0.92 [12] (predicate abstraction and thread-modular reasoning), and SATABS 3.1 [13] (SAT-based predicate abstraction).

To this end, we use the concurrency benchmarks from the *Second Competition on Software Verification* [14], which includes typical mutual exclusion protocols, such as Dekker, Peterson, Szymanski and Lamport, as well as programs that manipulate concurrent data structures.

Some benchmarks contain unbounded loops, which can be handled by IMPARA, SATABS and THREADER, while CBMC and ESBMC require an unwinding limit, which we set to 6, the maximum among the *bounded* loops. Partial loop exploration is marked with a star superscript at the respective running time.

We observe that IMPARA shows promising performance compared to the other tools, despite its prototype status. The running time for selected benchmarks are given in Table I. Each program contains assertions to be verified. Column "safe" indicates if the respective program is safe.

IMPARA 0.2 uses CBMC 4.5 as a front end. The back end, including the symbolic-execution engine, was written from scratch. To focus the implementation effort on the concurrency aspect, we use syntactic weakest preconditions as an interpolation procedure. For many typical concurrency benchmarks, weakest preconditions give sufficient invariants. However, we anticipate that leveraging a more advanced interpolation procedure could further improve performance.

We have implemented optimizations to speed up the frequently occurring cover checks. In a cascaded approach, we first use syntactic checks to cover trivial implications that can be resolved syntactically, e.g., $x > 0 \wedge y > 0$ trivially implies $y > 0$. Then we look up the implication in a table. Finally, if that fails, we invoke a SMT solver to check implication.

Benchmarks Using Weak Memory Consistency: The presented algorithm assumes interleaving semantics. Modern multi-core architectures, however, implement weaker consistency models, and therefore permit additional behaviors. Our technique can be extended to support popular consistency models including TSO (x86), PSO, RMO and PowerPC by combining it with the instrumentation proposed in [15].

The `sql` benchmark is a bug in PostgreSQL worker synchronization that occurs on the PowerPC architecture. A developer fix has also been found to be buggy. IMPARA is able to verify the safe programs and find counterexamples except for the PowerPC variant of the PostgreSQL benchmark where the tool times out. We anticipate that this can be fixed by a more aggressive expression simplification.

Effect of dPOR and Force Covers: To evaluate the benefit of dynamic partial-order reduction, and to compare different

program	safe	CBMC		ESBMC		SATABS		THREADER		IMPARA	
Sober benchmark											
SC	y	**0.3**	✓	1.2	✓	**0.3**	✓	120	FN	0.7	✓
TSO	n	**0.5**	✓	TO		2.6	✓	ERR		3.7	✓
RMO	n	**0.5**	✓	TO		2.5	✓	ERR		3.6	✓
PSO	n	**0.3**	✓	TO		1.4	✓	ERR		1.7	✓
Power	n	**0.3**	✓	TO		1.4	✓	ERR		1.7	✓
fix_SC	y	**0.3**	✓	1.3	✓	0.4	✓	120	FN	0.7	✓
fix_TSO	y	**0.3**	✓	TO		5.5	✓	ERR		1.3	✓
fix_PSO	y	**0.3**	✓	TO		5.6	✓	ERR		1.4	✓
fix_power	y	**0.3**	✓	TO		5.6	✓	ERR		1.4	✓
SQL benchmark											
SC	y	1.8*	(✓)	475.6*	✓	**0.3**	✓	1.7	FN	0.4	✓
TSO	y	6.9*	(✓)	TO		**0.3**	✓	3.25	FN	0.5	✓
Power	n	824.9*	✓	TO		TO		ERR		TO	
dev_fix_Power	n	TO		TO		17.7	FP	ERR		TO	

TABLE III: IMPARA on weak memory benchmarks

combinations of force cover and partial-order reduction, we experiment with four different configurations of IMPARA:

- **sPOR**: expands interleavings only when an action is executed that operates on shared variables; the original implementation of CLOSE is used.
- **sPOR+FC**: **sPOR** with force cover (FC).
- **dPOR**: dPOR-IMACT without force cover; this requires the $\text{CLOSE}_\Diamond$ function described in Sec. IV-C.
- **dPOR+FC**: **dPOR** with force cover.

Table II compares the four different configurations in terms of their running time ("s" for seconds), number of nodes ("$|V|$") and number of cover checks that require an implication check by an SMT solver ("impl"). Runs that have timed out are recorded with "TO" in the time field, and all other fields are filled with "–". To quantify the penalty incurred by cover expansions from $\text{CLOSE}_\Diamond$, we give the percentage ("C") of nodes resulting from cover expansions, e.g., 15% for `read_write_s` and around 27% for safe Sober weak-memory examples (to save space, we omit detailed results for weak memory benchmarks). Note that cover expansions were crucial to find assertion violations in the weak-memory benchmarks. For safe programs, we find that dPOR always produces less nodes than sPOR despite cover expansions.

Clearly, all configurations beat sPOR. On the other hand, we observe that POR and FC are complementary techniques. POR removes redundancies arising from thread interleaving, while FC covers thread-internal branching, e.g., from conditionals and loops, *as well as* redundant thread interleavings. For the latter, FC needs more unwindings than POR. For the smaller examples, these additional unwindings are few, as paths remain short, but the cost increases in larger programs. Comparing FC and POR, we observe that POR tends to reduce the number of necessary implications checks. This is because FC catches redundant interleavings that are removed by POR, and because it is a refinement technique, which triggers implication checks. Again, the cost of implication checks increases with program size, which can make POR scale better to larger problems.

VI. RELATED WORK

Partial-order reduction (POR) [2]–[4] has been proposed as a technique to combat state explosion by exploring only a representative subset of all possible interleavings, and has been implemented in the explicit-state model checkers SPIN [16] and Verisoft [5]. Dynamic POR techniques [17], [18] are based on the same concepts as classical static POR but capture dynamic

			sPOR			sPOR+FC			dPOR				dPOR+FC			
	LOC	safe	s	$\|V\|$	impl	s	$\|V\|$	impl	s	$\|V\|$	impl	C	s	$\|V\|$	impl	C
dekker	57	y	1.3	7407	9	0.2	504	8	0.1	433	3	0%	**0.1**	331	2	0%
lamport	79	y	9.6	36624	223	1.0	4740	226	1.0	7418	149	54%	**0.3**	1700	205	13%
peterson	45	y	0.7	3155	55	**0.1**	419	89	0.3	1081	43	0%	**0.1**	199	16	0%
szymanski	57	y	2.3	13332	14	0.3	1059	5	**0.2**	1264	5	0%	**0.2**	673	2	0%
read_write_u	59	n	5.4	31786	23	1.0	7628	119	1.6	18261	59	0%	**0.6**	4899	53	0%
read_write_s	68	y	75.2	109096	148	7.0	12932	1457	5.0	36777	129	14%	**0.9**	7065	223	15%
time_var_mutex	92	y	0.2	867	4	0.2	867	6	0.2	435	3	0%	**0.1**	252	1	0%
stack_u	144	n	TO	–	–	TO	–	–	3.1	2589	717	0%	**0.5**	424	81	0%
stack_s	144	y	TO	–	–	TO	–	–	TO	–	–	–	**38.8**	2037	4420	0%

TABLE II: Comparison of different configurations of IMPARA

dependencies induced by pointers on-the-fly during the state-space exploration.

Our monotone exploration strategy dPOR corresponds to the one used in [6], where POR is applied to SMT-based bounded model checking. The idea of cover expansions in function $\text{CLOSE}_\Diamond$ of our algorithm is inspired by a similar precaution in stateful dynamic POR [19].

Cimatti et al. combine static POR with lazy abstraction [20] to verify SystemC programs. There are several differences to our approach: our POR technique aims at dynamic dependencies induced by pointers, we are using Impact rather than predicate abstraction, and our approach is geared towards multi-threaded programs rather than SystemC programs.

Gupta et al. combine predicate abstractions with thread-modular proof rules [21], [22] in a tool called THREADER [23].

In the setting of single-threaded programs, the IMPACT algorithm has been re-implemented in a tool called WOLVERINE and compared with SATABS [24]. Beyer et al. have developed an approach where different invariant-generation techniques can be combined in a configurable tool CPA-CHECKER [25], together with techniques such as large block encoding [26]. Using CPA-CHECKER, they compare predicate abstraction with Impact [8] and evaluate the effectiveness of force covers.

VII. CONCLUSION

We have presented a new software model checking technique for concurrent programs based on lazy abstraction with interpolants and partial-order reduction, which performs very favorably compared to existing tools. In the future, we would like to incorporate more advanced invariant-generation techniques and investigate more aggressive POR techniques.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, and also Subodh Sharma, Luis María Ferrer Fioriti and Matt Lewis for their valuable feedback.

REFERENCES

[1] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.

[2] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems*, ser. LNCS. Springer, 1996, vol. 1032.

[3] D. Peled, "All from one, one for all: on model checking using representatives," in *CAV*, ser. LNCS, vol. 697. Springer, 1993, pp. 409–423.

[4] A. Valmari, "Stubborn sets for reduced state space generation," in *Applications and Theory of Petri Nets*, ser. LNCS, vol. 483. Springer, 1989, pp. 491–515.

[5] P. Godefroid, "Software model checking: The VeriSoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[6] C. Wang, Z. Yang, V. Kahlon, and A. Gupta, "Peephole partial order reduction," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 382–396.

[7] K. L. McMillan, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.

[8] D. Beyer and P. Wendler, "Algorithms for software model checking: Predicate abstraction vs. Impact," in *FMCAD*. IEEE, 2012, pp. 106–113.

[9] A. W. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets*, ser. LNCS, vol. 255. Springer, 1986, pp. 279–324.

[10] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *CAV*, 2013, pp. 141–157.

[11] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *ICSE*. ACM, 2011, pp. 331–340.

[12] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *CAV*, 2011.

[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT," *Formal Methods in System Design (FMSD)*, vol. 25, pp. 105–127, September–November 2004.

[14] D. Beyer, "Second competition on software verification – (summary of SV-COMP 2013)," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013, pp. 594–609.

[15] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, "Software verification for weak memory via program transformation," in *ESOP*, ser. LNCS, vol. 7792. Springer, 2013, pp. 512–532.

[16] G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[17] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*. ACM, 2005, pp. 110–121.

[18] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, "Cartesian partial-order reduction," in *SPIN*, ser. LNCS, vol. 4595. Springer, 2007, pp. 95–112.

[19] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction," in *SPIN*, ser. LNCS, vol. 5156. Springer, 2008, pp. 288–305.

[20] A. Cimatti, I. Narasamdya, and M. Roveri, "Boosting lazy abstraction for SystemC with partial order reduction," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 341–356.

[21] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, pp. 319–340, 1976.

[22] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, 1983.

[23] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*. ACM, 2011, pp. 331–344.

[24] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with WOLVERINE," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 573–578.

[25] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190.

[26] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *FMCAD*. IEEE, 2009, pp. 25–32.

Proving Termination of Imperative Programs Using Max-SMT

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, Albert Rubio
Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*—We show how Max-SMT can be exploited in constraint-based program termination proving. Thanks to expressing the generation of a ranking function as a Max-SMT optimization problem where constraints are assigned different weights, quasi-ranking functions –functions that almost satisfy all conditions for ensuring well-foundedness– are produced in a lack of ranking functions. By means of trace partitioning, this allows our method to progress in the termination analysis where other approaches would get stuck. Moreover, Max-SMT makes it easy to combine the process of building the termination argument with the usually necessary task of generating supporting invariants. The method has been implemented in a prototype that has successfully been tested on a wide set of programs.**

I. Introduction

Proving termination is necessary to ensure total correctness of programs. Still, termination bugs are difficult to trace and are hardly notified: as they do not arise as system failures but as unresponsive behavior, when faced to them users tend to restart their devices without reporting to software developers. Due to this, approaches for proving termination of imperative programs have regained an increasing interest in the last decade [1]–[4].

One of the major difficulties in these methods is that often *supporting invariants* are needed. E.g., in [5] linear invariants are exhaustively computed before termination analysis. In the same paper a heuristic approach is also presented, which only requires a light-weight invariant generator by restricting to single-variable ranking functions. Another solution is proposed in [6], where invariant generation is not performed eagerly but on demand. By formulating both invariant and ranking function synthesis as constraint problems, both can be solved simultaneously, so that only the necessary supporting invariants for the targeted ranking functions –namely, *lexicographic linear ranking functions*– need to be discovered.

Based on [5], [6], we present a Max-SMT constraint-based approach for proving termination. The crucial observation in our method is that, albeit our goal is to show that transitions cannot be executed infinitely by finding a ranking function or an invariant that disables them, if we only discover an invariant, or an invariant and a *quasi-ranking function* that almost fulfills all needed properties for well-foundedness, we have made some progress: either we can remove *part of a transition* and/or we have improved our knowledge on the behavior of the program. A natural way to implement this idea is by considering that some of the constraints are *hard* (the ones guaranteeing invariance) and others are *soft* (those guaranteeing well-foundedness) in a Max-SMT framework. Moreover, by giving different weights to the constraints we can set priorities and favor those invariants and (quasi-) ranking functions that lead to the furthest progress.

The technique has been implemented in our tool CppInv, which analyses programs with integer variables and linear expressions. Thanks to it, we have proved termination of a wide set of programs, which have been taken from the programming learning environment Jutge.org [7] and from benchmark suites in the literature [8].

A. Related Work.

As mentioned above, our method is based on [5]. Namely, we have borrowed the core argument for termination proofs, which is based on iteratively discarding those transitions that cannot be executed infinitely. However, we improve on the way supporting invariants are generated. While in [5] invariants are pre-computed in a process that is independent from the termination analysis and which turns out to be the bottleneck of the approach, we find lazily the invariants needed to ensure that ranking functions meet their requirements.

Our research also builds upon [6], where the constraint-based method [9] was first applied to termination. However, we extend this work in several aspects. First, in that approach only linear programs with unnested loops can be handled, while we can deal with arbitrary control structures. Moreover, in [6] the generation of their lexicographic ranking functions requires a higher-level loop that, before sending the constraint problem to the solver, determines the precedence of the transitions in the lexicographic order. On the other hand, in our approach this outer loop is not needed. Finally, thanks to assigning weights to the constraints, unlike [6] we do not need to stipulate the number of supporting invariants that will be needed a priori, and hence our constraint problems are simpler. Further, weights allow us to guide the solving engine in the search of appropriate ranking functions and invariants.

In [10], the lexicographic approach of [6] is extended so as to handle programs with complex control flow. However, their method still requires to search for the right ordering of the transitions in order to obtain a successful termination proof. Moreover, in this technique the procedures for synthesizing ranking functions and auxiliary invariants do not share enough information, while in our proposal these mechanisms are tightly coupled. Finally, in [8] a method closely related to ours is presented. Both approaches, which have been developed independently, go in the same direction of achieving a better cooperation between the invariant and the ranking function

ISBN 978-0-9835678-3-7/13.

syntheses. Still, a significant difference is that we can exploit the quasi-ranking functions produced in the absence of ranking functions in order to progress in the termination analysis.

In addition to lexicographic ranking functions, there is a group of effective tools whose termination arguments are based on Ramsey's Theorem and the notion of *transition invariant* [11]. Transition invariants are over-approximations of the transitive closure of the transition relation restricted to the reachable state space. The crucial observation is that a transitive relation that is *disjunctively well-founded*, i.e., that is included in the union of well-founded relations, must be well-founded too. Hence, if one is able to find a transition invariant that is also disjunctively well-founded, the program must be terminating. In [12], this transition invariant is computed iteratively, starting from the empty relation, by discovering unranked paths of the program thanks to a reachability check, and using the approach in [3] for synthesizing new ranking functions for them. On the other hand, in [13] the generation of the disjunctively well-founded transition invariant is performed bottom-up from innermost loops by identifying invariant and transitive relations among a set of templates that are disjunctively well-founded by construction. Nested loops are then handled thanks to loop summarization. Our techniques can also be seen as producing a disjunctively well-founded transition invariant, being the difference with respect to the previous approaches in the way new unranked paths are identified and how a termination argument is generated for them.

Finally, a problem related to proving termination that has recently raised interest in the area is that of *conditional termination:* to synthesize automatically preconditions on the inputs that ensure program termination. In this context, in [15] the authors consider what they call *potential ranking functions*, which are functions over program states that are bounded but not necessarily decreasing. The quasi-ranking functions that we consider here are more general, as for instance functions that are decreasing but not bounded are also included. In [16], the problem of conditional termination is also considered. The approach is based on disjunctively well-founded relations as in [12], but instead of identifying unranked program paths, thanks to a dual inclusion the authors partition the transition relation into those behaviors already proved to be terminating and those whose status is still unknown. In our work we also proceed by splitting the transition relation into a terminating part and an unknown part. However, in [16] this division is achieved by means of a fixpoint computation, while our approach is constraint-based.

II. Preliminaries

A. SMT and Max-SMT

Let $\mathcal{P}$ be a finite set of *propositional variables*. If $p \in \mathcal{P}$, then p and $\neg p$ are *literals*. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals. A *propositional formula* is a conjunction of clauses. The problem of *propositional satisfiability* (abbreviated as SAT) consists in, given a formula, to determine whether or not it is *satisfiable*, i.e., if it has a *model*: an assignment of Boolean values to variables that satisfies the formula.

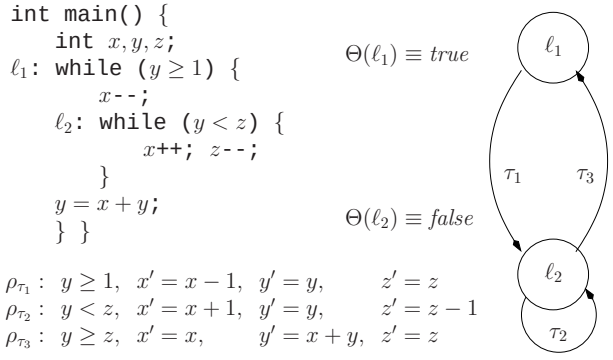


Fig. 1. Program and its transition system.

An extension of SAT is the *satisfiability modulo theories (SMT)* problem [17]: to decide the satisfiability of a given quantifier-free first-order formula with respect to a background theory. Here we will consider the theories of *linear arithmetic (LA)*, where literals are linear inequalities, and the more general theory of *non-linear arithmetic (NA)*, where literals are polynomial inequalities.

Another generalization of SAT is the *Max-SAT* problem [17]: it consists in, given a *weighted* formula F where each clause C_i has a weight ω_i (a positive number or infinity), to find the assignment such that the cost, i.e., the sum of the weights of the falsified clauses, is minimized. Clauses with infinite weight are called *hard*, while the rest are called *soft*. Equivalently, the problem can be seen as finding the model of the hard clauses such that the sum of the weights of the falsified soft clauses is minimized.

Finally, the problem of *Max-SMT* [18] merges Max-SAT and SMT, and is defined from SMT analogously to how Max-SAT is derived from SAT. Namely, the *Max-SMT* problem consists in, given a weighted formula, to find an assignment that minimizes the sum of the weights of the falsified clauses in the background theory.

B. Transition Systems, Invariants and Ranking Functions

Henceforth we will model imperative programs by means of *transition systems*. A transition system $\mathcal{S} = (\overline{v}, \mathcal{L}, \Theta, \mathcal{T})$ consists of a tuple of *variables* $\overline{v}$, a set of *locations* $\mathcal{L}$, a map Θ from locations to formulas characterizing the initial values of the variables, and a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a triple (ℓ, ℓ', ρ), where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and ρ is the *transition relation*: a formula over the program variables $\overline{v}$ and their primed versions $\overline{v}'$, which represent the values of the variables after the transition. See Fig. 1 for an example of a program together with a corresponding representation as a transition system.

From now on we assume that variables take *integer* values and programs are *linear*, i.e., the initial conditions Θ and transition relations ρ are described as conjunctions of linear inequalities. Strict inequalities may be translated into non-strict ones thanks to the integer type of the variables.

A *state* is an assignment of a value to each of the variables in $\overline{v}$. A *configuration* is a pair (ℓ, σ) consisting of a location ℓ and a state σ. A *computation* is a (possibly infinite) sequence of configurations (ℓ_0, σ_0), (ℓ_1, σ_1), ... such that $\sigma_0 \models \Theta(\ell_0)$, and for each pair of consecutive configurations (ℓ_i, σ_i) and $(\ell_{i+1}, \sigma_{i+1})$, there exists a transition $\tau = (\ell_i, \ell_{i+1}, \rho) \in \mathcal{T}$ such that $(\sigma_i, \sigma_{i+1}) \models \rho$. A configuration (ℓ, σ) is *reachable* if there exists a computation ending at (ℓ, σ). A transition system is said to be *terminating* if all its computations are finite. The problem that we target in this work is, given a transition system, to determine if it is terminating or not.

A transition $\tau = (\ell, \ell', \rho)$ is *disabled* if it can never be executed, i.e., if for all reachable configuration (ℓ, σ), there does not exist any σ' such that $(\sigma, \sigma') \models \rho$. A transition τ is called *finitely executable* if in any computation, τ is only executed a finite number of times (in particular, if τ is disabled). Otherwise, i.e., if there exists a computation where τ is executed infinitely, we say that τ is *infinitely executable*.

An *assertion* is a first-order formula over $\overline{v}$. An assertion I is an *invariant* at location ℓ if for any reachable configuration (ℓ, σ), it holds that $\sigma \models I$. An *invariant map* μ assigns an invariant $\mu(\ell)$ to each of the locations ℓ. An important class of invariant maps is that of *inductive invariant maps*:

Definition 1: An invariant map μ is said to be *inductive* if:

- **[Initiation]** For every location $\ell \in \mathcal{L}$: $\Theta(\ell) \models \mu(\ell)$
- **[Consecution]** For every transition $\tau = (\ell, \ell', \rho) \in \mathcal{T}$: $\mu(\ell) \wedge \rho \models \mu(\ell')'$.

Invariant maps are fundamental when analyzing program termination. For instance, a transition $\tau = (\ell, \ell', \rho)$ is proved to be disabled if there is an invariant $\mu(\ell)$ at location ℓ such that $\mu(\ell) \wedge \rho$ is unsatisfiable. In general, if μ is an invariant map, then any transition $\tau = (\ell, \ell', \rho)$ can be safely strengthened by replacing the transition relation ρ by $\mu(\ell) \wedge \rho$.

The basic idea of the approach we follow for proving program termination [5] is to argue by contradiction that no transition is infinitely executable. First of all, no disabled transition can be infinitely executable trivially. Moreover, one just needs to focus on transitions joining locations in the same strongly connected component (SCC): if a transition is executed over and over again, then its pre and post locations must belong to the same SCC. So let us assume that one has found a *ranking function* for such a transition τ, according to:

Definition 2: Let $\tau = (\ell, \ell', \rho)$ be a transition such that ℓ and ℓ' belong to the same SCC, denoted by C. A function $R : \overline{v} \to \mathbb{Z}$ is said to be a *ranking function* for τ if:

- **[Boundedness]** $\rho \models R \geq 0$
- **[Strict Decrease]** $\rho \models R > R'$
- **[Non-increase]** For every $\hat{\tau} = (\hat{\ell}, \hat{\ell}', \hat{\rho}) \in \mathcal{T}$ such that $\hat{\ell}, \hat{\ell}' \in C$: $\hat{\rho} \models R \geq R'$

Note that boundedness and strict decrease *only* depend on τ, while non-increase depends on *all* transitions in the SCC.

The key result is that if $\tau = (\ell, \ell', \rho)$ admits a ranking function R, then it is finitely executable. Indeed, first notice that if one can execute τ from a configuration (ℓ, σ) then $R(\sigma) \geq 0$, because of boundedness. Also, the value of R at the states along any path contained in C cannot increase, thanks to the non-increase property. Moreover, in any cycle contained in C traversing τ, the value of R strictly decreases, due to the strict decrease property. Now, let us assume that there was a computation where τ was executed infinitely. Such a computation would eventually visit only locations in C. Because of the previous observations, by evaluating R at the states at which τ is executed we could construct an infinitely decreasing sequence of non-negative integers, a contradiction.

Finitely executable transitions can be safely removed from the transition system as regards termination analysis. This in turn may break the SCC's into smaller pieces. If by applying this reasoning recursively one can prove that all transitions are finitely executable, then the transition system is terminating.

C. Constraint-Based Program Analysis

Here we review the *constraint-based program analysis* approach [6], [9]. The idea is to consider a template for candidate invariant properties (respectively, ranking functions), e.g., linear inequalities (linear expressions). These templates involve both program variables as well as unknowns whose values have to be determined so as to ensure the required properties. To this end, the implications in Definition 1 (Definition 2) are expressed by means of *constraints* (hence the name of the approach) on the unknowns. If implications are encoded soundly, any solution to the constraints yields an invariant map (ranking function). Specifically, if linear arithmetic is the target language, this can be achieved with Farkas' Lemma:

Theorem 1 (Farkas' Lemma): Let S be a system of linear inequalities $Ax + b \leq 0$ ($A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$) over real variables $x^T = (x_1, \ldots, x_n)$. When S is satisfiable, it entails a linear inequality $c^T x + d \leq 0$ ($c \in \mathbb{R}^n, d \in \mathbb{R}$) iff there is $\lambda \in \mathbb{R}^m$ such that $\lambda \geq 0$, $c^T = \lambda^T A$ and $d \leq \lambda^T b$. Further, S is unsatisfiable iff $1 \leq 0$ can be so derived.

For clarity, henceforth the following notation is used. Given a conjunction of linear inequalities $Ax + b \leq 0$ and a linear inequality $c^T x + d \leq 0$, where the coefficients a_{ij}, b_i, c_j, d may be real numbers or unknowns, we denote by $Ax + b \leq 0 \vdash c^T x + d \leq 0$ the set of constraints on the unknown coefficients and on fresh real unknowns $\lambda = (\lambda_1, \ldots, \lambda_m)$, consisting in $\lambda \geq 0$, $c^T = \lambda^T A$ and $d \leq \lambda^T b$.

III. Termination Analysis with Max-SMT

In this section we first describe a constraint-based method for termination analysis that uses SMT and identify some of its shortcomings (Sect. III-A). Then we show how Max-SMT can be used to overcome these limitations (Sect. III-B).

A. An SMT Approach to Proving Termination

Following the approach described in Sect. II-B [5], to show that a transition τ is finitely executable and thus discard it, one needs either a disability argument or a ranking function for it. To this end we construct a constraint system, i.e. an SMT formula, whose solutions correspond to either an invariant that proves disability, or a ranking function. Given an SCC, the constraint system, if satisfiable, will allow discarding (at least,

but possibly more than) one of the transitions in the SCC. By iterating this procedure until no cycles are left we will obtain a termination argument for the SCC.

To construct the constraint system, first of all we consider:

- for each location ℓ, a linear invariant template $I_\ell(\overline{v}) \equiv i_{\ell,0} + \sum_{v\in\overline{v}} i_{\ell,v}\cdot v \leq 0$, where $i_{\ell,0}$, $i_{\ell,v}$ are unknown;
- a linear ranking function template $R(\overline{v}) \equiv r_0 + \sum_{v\in\overline{v}} r_v \cdot v$, where r_0, r_v are unknown.

Recall that ranking functions are associated to transitions, not to locations. However, instead of introducing a template for each transition, we just have one single template, which, if the constraint system has a solution, will be a ranking function for a transition *to be determined by the solver*.

Similarly to [6], we take the following constraints from the definitions of inductive invariant and ranking function:

Initiation:	For $\ell \in \mathcal{L}$:	$\mathbb{I}_\ell \stackrel{def}{=} \Theta(\ell) \vdash I_\ell$
Disability:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{D}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash 1 \leq 0$
Consecution:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{C}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash I'_{\ell'}$
Boundedness:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{B}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq 0$
Strict Decrease:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{S}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R > R'$
Non-increase:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{N}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq R'$

Let L and T be the sets of locations and transitions in the SCC in hand, respectively. Let also P be the set of *p*ending transitions, i.e., which have not been proved to be finitely executable yet. Then we build the next constraint system:

$$\bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge \bigvee_{\tau\in P} (\mathbb{D}_\tau \vee (\mathbb{B}_\tau \wedge \mathbb{S}_\tau)) \wedge ((\bigwedge_{\tau\in P} \mathbb{N}_\tau) \vee \bigvee_{\tau\in P} \mathbb{D}_\tau).$$

The first two conjuncts guarantee that an invariant map is computed; the other two, that at least one of the pending transitions can be discarded. Notice that, if there is no disabled transition, we ask that *all* transitions in P are non-increasing, but only that at least *one* transition in P (the next to be removed) is both bounded and strict decreasing. Note also that for finding invariants one has to take into account *all* transitions in the SCC, even those that have already been proved to be finitely executable: otherwise some reachable states might not be covered, and the invariant generation would become unsound. Hence in our termination analysis we consider two transition systems: the original transition system for invariant synthesis, whose transitions are T and which remains all the time the same; and the *termination transition system*, whose transitions are P, i.e, where transitions already shown to be finitely executable have been removed. This duplication is similar to the *cooperation graph* of [8].

However, this first approach is problematic when a ranking function needs several invariants. A possible solution is to add more templates iteratively, so that for example initially invariants consisting of a single linear inequality are tried, if unsuccessful then invariants consisting of a conjunction of two linear inequalities are tried, etc. But when proceeding in this way, all problems before the right number of invariants is found are unsatisfiable. This is wasteful, as no constructive information is retrieved from unsatisfiable constraint systems.

Another problem with this method for analyzing termination is that the kind of termination proofs it yields may be too restricted. More specifically, when one proves that a transition τ is finitely executable, then a single termination argument shows there is no computation where τ appears infinitely. Although this produces compact proofs, on the other hand sometimes there may not exist such a unique reason for termination, and it becomes necessary a more fine-grained examination. However, the approach as presented so far does not provide a natural way or guidance for refining the analysis.

B. A Max-SMT Approach to Proving Termination

The main contribution of our work is to show that the constraint system can be expressed in such a way that, even when it turns out to be unsatisfiable, some information useful for refining the termination analysis can be obtained. The key observation is that, even though our aim is to prove transitions to be finitely executable (by finding a ranking function or an invariant that disables them), if we just find an invariant, or an invariant and a *quasi-ranking function* that is close to fulfill all required conditions, we have progressed in our analysis.

The idea is to consider the constraints guaranteeing invariance as *hard*, so that any solution to the constraint system will satisfy them, while the rest are *soft*. Let us consider propositional variables $p_\mathbb{B}$, $p_\mathbb{S}$ and $p_\mathbb{N}$, which intuitively represent if the conditions of boundedness, strict decrease and non-increase in the definition of ranking function are violated respectively, and corresponding weights $\omega_\mathbb{B}$, $\omega_\mathbb{S}$ and $\omega_\mathbb{N}$. We consider now the next constraint system (where soft constraints are written $[\cdot,\omega]$, and hard ones as usual):

$$\bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} \Big(\mathbb{D}_\tau \vee \mathbb{C}_\tau\Big) \wedge \bigvee_{\tau\in P} \Big(\mathbb{D}_\tau \vee \big((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S})\big)\Big) \wedge$$
$$\Big((\bigwedge_{\tau\in P} \mathbb{N}_\tau) \vee \bigvee_{\tau\in P} \mathbb{D}_\tau \vee p_\mathbb{N}\Big) \wedge [\neg p_\mathbb{B}, \omega_\mathbb{B}] \wedge [\neg p_\mathbb{S}, \omega_\mathbb{S}] \wedge [\neg p_\mathbb{N}, \omega_\mathbb{N}].$$

Note that ranking functions have cost 0, and (if no transition is disabled) functions that fail in any of the conditions are penalized with the respective weight. Thus, the Max-SMT solver looks for the best solution and gets a ranking function if feasible; otherwise, the weights guide the search to get invariants and quasi-ranking functions that satisfy as many conditions as possible.

Hence this Max-SMT approach allows recovering information even from problems that would be unsatisfiable in the initial method. This information can be exploited to perform dynamic trace partitioning [19] as follows. Assume that the optimal solution to the above Max-SMT formula has been computed, and let us consider a transition $\tau \in P$ such that $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ evaluates to true in the solution. Then we distinguish several cases depending on the properties satisfied by τ and the computed function R:

- If τ is disabled then it can be removed.
- If R is non-increasing and satisfies boundedness and strict decrease for τ, then τ can be removed too: R is a ranking function for it.
- If R is non-increasing and satisfies boundedness for τ but not strict decrease, one can split τ in the termination

transition system into two new transitions: one where $R > R'$ is added to τ, and another one where $R = R'$ is enforced. Then the new transition with $R > R'$ is automatically eliminated, as R is a ranking function for it. Equivalently, this can be seen as adding $R = R'$ to τ. Now, if the solver could not prove R to be a true ranking function for τ because it was missing an invariant, this transformation will guide the solver to find that invariant so as to disable the transition with $R = R'$.

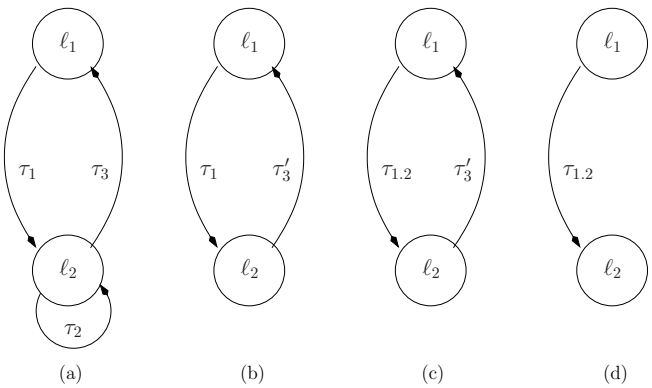
- If R is non-increasing and satisfies strict decrease for τ but not boundedness, the same technique from above can be applied: it boils down to adding $R < 0$ to τ.
- If R is non-increasing but neither strict decrease nor boundedness are fulfilled for τ, then τ can be split into two new transitions: one with $R < 0$, and another one with $R \geq 0 \wedge R = R'$.
- If R does not satisfy the non-increase property, then it is rejected; however, the invariant map from the solution can be used to strengthen the transition relations for the following iterations of the termination analysis.

Note this analysis may be worth applying on other transitions τ in the termination transition system apart from those that make $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ true. E.g., if R is a ranking function for a transition τ but fails to be so for another one τ' because strict decrease does not hold, then, according to the above discussion, τ' can be strengthened with $R = R'$.

On the other hand, working in this iterative way requires imposing additional constraints to avoid getting to a standstill. Namely, in the case where non-increase does not hold and so one would like to exploit the invariant, it is necessary to impose that the invariant is not redundant. More in detail, let us consider a fixed location ℓ, and let $I_\ell^{(1)}, \ldots, I_\ell^{(k)}$ be the previously computed invariants at location ℓ. Then I_ℓ, the invariant to be generated at ℓ, is redundant if it is implied by $I_\ell^{(1)}$, ..., $I_\ell^{(k)}$, i.e., if $\mathbb{E}_\ell \stackrel{def}{=} \forall \overline{v}\ (I_\ell^{(1)}(\overline{v}) \wedge \ldots \wedge I_\ell^{(k)}(\overline{v}) \rightarrow I_\ell(\overline{v}))$. So we impose $p_\mathbb{N} \rightarrow \neg \bigwedge_{\ell \in L} \mathbb{E}_\ell$ to ensure that violating non-increase leads to non-redundant invariants. Conditions are added similarly to avoid redundant quasi-ranking functions.

Another advantage of this Max-SMT approach is that by using different weights we can express priorities over conditions. Since, as explained above, violating the property of non-increase invalidates the computed function R, it is convenient to make $\omega_\mathbb{N}$ the largest weight. On the other hand, when non-increase and boundedness are fulfilled but not strict decrease an equality is added to the transition, whereas when non-increase and strict decrease are fulfilled but not boundedness just an inequality is added. As we prefer the former to the latter, in our implementation (see Sect. V) we set $\omega_\mathbb{B} > \omega_\mathbb{S}$.

A further improvement is the generation of *termination implications*. A termination implication at a location ℓ is an assertion $J(\overline{v})$ such that any transition in the *termination transition system* that leads into ℓ implies it, i.e., it holds that $\rho \models J(\overline{v}')$, where ρ is the relation of the transition. Thus, J will *eventually* hold when ℓ is reached (although, unlike ordinary invariants, may not initially be true; see Example 1 below). Hence, it can be propagated forward in the termination transition system to the transitions going out from ℓ. To produce termination implications, for each location ℓ a new linear inequality template $J_\ell(\overline{v})$ is introduced and the following constraint is imposed: $\bigwedge_{\tau=(\hat{\ell},\ell,\rho)\in P}(\mathbb{D}_\tau \vee I_{\hat{\ell}} \wedge \rho \vdash J'_\ell)$. Additional constraints are enforced to ensure that new termination implications are not redundant with the already computed invariants and termination implications.



$\Theta(\ell_1) \equiv \mathit{true} \qquad \Theta(\ell_2) \equiv \mathit{false}$

$$
\begin{array}{lllll}
\rho_{\tau_1}: & & y \geq 1,\ x' = x-1,\ y' = y, & & z' = z \\
\rho_{\tau_{1.2}}: & x < 0,\ & y \geq 1,\ x' = x-1,\ y' = y, & & z' = z \\
\rho_{\tau_2}: & & y < z,\ x' = x+1,\ y' = y, & & z' = z-1 \\
\rho_{\tau_3}: & & y \geq z,\ x' = x, & y' = x+y, & z' = z \\
\rho_{\tau'_3}: & y \geq 1,\ & y \geq z,\ x' = x, & y' = x+y, & z' = z
\end{array}
$$

Fig. 2. Evolution of the termination transition system: initially (a) and after the first (b), second (c) and third (d) round.

Example 1: Let us show a termination analysis of the program in Fig. 1. In the first round, the solver finds the invariant $y \geq 1$ at ℓ_2 and the ranking function z for τ_2. While $y \geq 1$ can be added to τ_3 (resulting into a new transition τ'_3), the ranking function allows eliminating τ_2 from the termination transition system (see Fig. 2 (b)).

In the second round, the solver cannot find a ranking function. However, thanks to the Max-SMT formulation, it can produce the quasi-ranking function x, which is non-increasing and strict decreasing for τ_1, but not bounded. This quasi-ranking function can be used to split transition τ_1 into two new transitions $\tau_{1.1}$ and $\tau_{1.2}$ as follows:

$$
\begin{array}{lllll}
\rho_{\tau_{1.1}}: & \boldsymbol{x \geq 0}, & y \geq 1, & x' = x-1, & y' = y, \quad z' = z \\
\rho_{\tau_{1.2}}: & \boldsymbol{x < 0}, & y \geq 1, & x' = x-1, & y' = y, \quad z' = z
\end{array}
$$

Then $\tau_{1.1}$ is immediately removed, since x is a ranking function for it. The current termination transition system is given in Fig. 2 (c).

In the third and final round, the termination implication $x < 0$ is generated at ℓ_2, together with the ranking function y for transition τ'_3. Note that the termination implication is crucial to prove the strict decrease of y for τ'_3, and that the previously generated invariant $y \geq 1$ at ℓ_2 is needed to ensure boundedness. Now τ'_3 can be removed, which makes the graph acyclic (see Fig. 2 (d)). This concludes the termination proof.

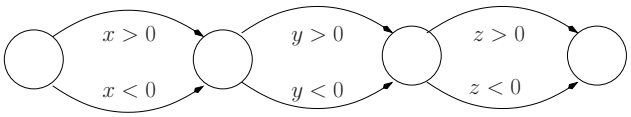


Fig. 3. Chain of locations obtained from a sequence of statements assume($x \neq 0$); assume($y \neq 0$); assume($z \neq 0$). Note disequalities are not natively supported, and so have to be split into disjunctions of inequalities.

IV. Implementation

The method presented in Sect. III has been implemented in the tool CppInv[1]. This section describes this implementation.

CppInv admits code written in C++ as well as in the language of T2 [10]. The system analyses programs with integer variables, linear expressions and function calls. Variables of other data types, such as floating-point variables, are treated as unknown values. Function calls are handled with techniques similar to those in [20], although currently the returned value is ignored. Further, for recursive functions, after a function call we assign unknowns to all variables that can be modified in the call (i.e., global variables and variables passed by reference).

In the transformation from the source code to the internal transition system representation, CppInv attempts to reduce the number of locations by composing transitions. Still, this preprocessing may result in an exponential growth in the number of transitions. As our technique does not require minimized transition systems for soundness, the tool stops this location minimization if a threshold number of transitions is reached. Moreover, whenever a chain of locations connected by transitions that do not modify variables (see Fig. 3) is detected, CppInv does not attempt to eliminate the locations: since no variable is updated, in these transitions any function satisfies the non-increase condition, while no ranking function is possible. For this reason, when producing the constraints, these transitions are ignored as far as termination is concerned, and are only considered for the generation of invariants.

Once the input is represented as a transition system, the actual termination analysis starts. See function *proved_TS_term*:

```
bool proved_TS_term(Trans_Sys S = (v, L, Θ, T)) {
  // C is the list of SCC's topologically sorted according to ordering ≺
  (C, ≺) = compute_SCCs_and_topologically_sort(S);
  for (C ∈ C by ≺) {
    (L, T) = (locations(C), transitions(C));
    P = copy(T);
    for (ℓ ∈ L : ∃(ℓ̂, ℓ, ρ) ∈ T with ℓ̂ ∈ Ĉ ≺ C)
      Θ(ℓ) = Θ(ℓ) ∨ SPost(ρ);
    if (not proved_SCC_term(L, T, P)) return false; }
  return true; }
```

The SCC's are computed and topologically sorted, and each SCC is processed according to this order. Processing an SCC involves first performing a copy of the transitions for keeping track of those not proven finitely executable yet. Then the initial conditions are updated with the strongest postconditions of the incoming transitions from previous SCC's, where the strongest postcondition of a transition relation $\rho(\overline{v}, \overline{v}')$ is the assertion $\mathrm{SPost}(\rho)(\overline{v}) \equiv \exists\, \overline{w}\ \ \rho(\overline{w}, \overline{v})$. Finally the SCC is analysed for termination. If it could not be proved terminating, the procedure stops. Otherwise the next SCC is dealt with.

The analysis of termination of SCC's is orchestrated by the function *proved_SCC_term*:

```
bool proved_SCC_term(Set_Loc L, Set_Trans T, Set_Trans P) {
  if ( dis_trans (L, T, P) or rank_fun(L, T, P) or term_impl(L, T, P)){
    if (P == ∅) return true;
    for (C′ SCC in the graph of P) {
      T′ = transitions(C′);
      if (T′ ≠ ∅ and not proved_SCC_term(L, T, T′)) return false; }
    return true; }
  else return false ; }
```

It takes as arguments: a set of locations L and a set of transitions T, corresponding to an SCC of the transition system; and the *termination transition system*: a non-empty set $P \subseteq T$ of transitions that still have to be proved finitely executable. As explained in Sect. II-B, one may assume that the graph induced by P is strongly connected. The function returns **true** if all transitions in P can be proved finitely executable. We found out that, instead of directly solving the full constraint system introduced in Sect. III-B, in practice it is preferable to proceed by phases. Each phase[2] (functions *dis_trans*, *rank_fun* and *term_imp*) attempts to remove transitions from P by different means, and returns **true** if P has become empty or it is no longer strongly connected. In the former case, we are done. In the latter, the same procedure is recursively called. If after all phases P is non-empty, we report failure to prove termination.

In the first phase (function *dis_trans*), CppInv attempts to eliminate transitions with disability arguments by generating the appropriate invariants (neither ranking functions nor termination implications are considered at this point). This is achieved by solving the following Max-SMT formula: $\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge (\bigvee_{\tau \in T} \mathbb{D}_\tau \vee p_{\mathbb{D}}) \wedge [\neg p_{\mathbb{D}}, \omega_{\mathbb{D}}]$[3], where $p_{\mathbb{D}}$ is a propositional variable meaning that no transition can be disabled, and $\omega_{\mathbb{D}}$ is the corresponding weight. Transitions that are detected to be disabled (by means of a call to an SMT solver) are removed both from the original and the termination transition system. Invariants are used to strengthen the transition relations as described in Sect. II-B. The process is repeated while new transitions can be disabled.

```
bool dis_trans (Set_Loc L, Set_Trans T, Set_Trans P) {
  cont = true;
  while (cont) {
    cont = false;
    for (τ = (ℓ, ℓ′, ρ) ∈ P)
      if (ρ is UNSAT) // τ is disabled
        (T, P) = (T − {τ}, P − {τ});
    if (P == ∅) return true;
    H = ⋀_{ℓ∈L} 𝕀_ℓ ∧ ⋀_{τ∈T} (𝔻_τ ∨ ℂ_τ) ∧ ⋁_{τ∈T} (𝔻_τ ∨ p_𝔻);
    S = [¬p_𝔻, ω_𝔻];
    (I, c) = solve(H ∧ S);        // I invariant map, c cost of solution
    if (c == ∞) break;            // No solution to hard clauses
    for (ℓ ∈ L, (ℓ, ℓ′, ρ) ∈ T) // Strengthen relation with invariant
      ρ = ρ ∧ I(ℓ);
    if (c == 0) cont = true; }
  return not is_strongly_connected (P); }
```

[1] CppInv, together with all benchmarks used in the experimental evaluation of Sect. V, is available at www.lsi.upc.edu/~albert/cppinv-term-bin.tar.gz.

[2] These phases have a time limit in our implementation although this is not made explicit in the pseudo-code shown below.

[3] Constraints that avoid redundancy are not included for simplicity.

In the second phase (function *rank_fun*), the system eliminates transitions by using ranking functions as arguments (termination implications are not considered at this point). If the computed function R satisfies the non-increase property, then each of the transitions τ in the termination transition system is examined and either removed if R is a ranking function for τ, or split otherwise, as described in Sect. III-B.

bool *rank_fun*(*Set_Loc* L, *Set_Trans* T, *Set_Trans* P){
  while (**true**) {
    $H = \bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} \mathbb{C}_\tau \wedge \bigvee_{\tau \in P} ((\mathbb{B}_\tau \vee p_{\mathbb{B}}) \wedge (\mathbb{S}_\tau \vee p_{\mathbb{S}})) \wedge \bigwedge_{\tau \in P} (\mathbb{N}_\tau \vee p_{\mathbb{N}})$
    $S = [\neg p_{\mathbb{B}}, \omega_{\mathbb{B}}] \wedge [\neg p_{\mathbb{S}}, \omega_{\mathbb{S}}] \wedge [\neg p_{\mathbb{N}}, \omega_{\mathbb{N}}]$;
    (I, R, c) = *solve*$(H \wedge S)$;
    if $(c == \infty)$ **return false**; // No solution to hard clauses
    for $(\ell \in L, (\ell, \ell', \rho) \in \mathcal{T})$ // Strengthen relation with invariant
      $\rho = \rho \wedge I(\ell)$
    for $(\tau = (\ell, \ell', \rho) \in P)$
      if (ρ *is* UNSAT) // τ is disabled
        $(T, P) = (T - \{\tau\}, P - \{\tau\})$;
    if (*non_increase*(R))
      for $(\tau \in P)$
        if (*bounded*(τ, R) **and** *strict_decrease*(τ, R)) $P = P - \{\tau\}$;
        else *split* (τ, R, P); // Splits τ
    if $(P == \emptyset$ **or not** *is_strongly_connected*$(P))$ **return true**; } }

The third and final phase (function *term_impl*, not detailed here for lack of space) is very similar to the previous one, with the difference that termination implications are also included.

As regards the constraints, we restrain ourselves to invariants and ranking functions with *integer* coefficients, since this allows us to exploit efficient non-linear solving techniques [21]. Moreover, in order to perform integer reasoning, the following variation of Farkas' Lemma, based on the Gomory-Chvátal cutting plane rule [22], is employed:

Lemma 1: Let $Ax+b \leq 0$ ($A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$) be a system of linear inequalities over integer variables $x^T = (x_1, \ldots, x_n)$, and $c^T x+d \leq 0$ ($c \in \mathbb{Z}^n, d \in \mathbb{R}$) be a linear inequality. If there is $\lambda \in \mathbb{R}^m$, $i \in \mathbb{Z}$ and $f \in \mathbb{R}$ such that $\lambda \geq 0$, $c^T = \lambda^T A$, $\lambda^T b = i - f$, $0 \leq f < 1$ and $i \geq d$, then $Ax + b \leq 0$ entails $c^T x + d \leq 0$.

Lemma 1 allows transforming an $\exists\forall$ problem into an $\exists$ problem. If all coefficients in the premise are known values, the resulting satisfiability problem is an SMT problem over LA. Otherwise, an SMT problem over NA is obtained. Furthermore, as some unknowns are integer (the coefficients) and some are real (the multipliers), the resulting problems have mixed types.

CppInv uses Barcelogic [23] for solving the generated constraints. The Max-SMT(NA) solver for mixed non-linear arithmetic in Barcelogic extends the techniques presented in [21] for solving SMT(NIA) problems. This is achieved by allowing integer and real variables in the underlying linear arithmetic solver, and wrapping this solver with a branch-and-bound scheme for optimization [18].

V. Experimental Evaluation

In this section we show experiments that evaluate the performance of CppInv on a wide set of examples, which have been taken from the online programming learning environment Jutge.org [7] (see www.jutge.org), and from benchmark suites in [8] and in research.microsoft.com/en-us/projects/t2/. We provide here a comparison with the new version of T2, which according to the results given in [8] is performing much better when proving termination than most of the existing tools, including Terminator [12], AProVE [25] or ARMC [24], among others. We have also tried CProver [13] and Loopfrog [14], but the results were not good on these sets of benchmarks. All experiments were performed on an Intel Core i7 with 3.40GHz clock speed and 16 GB of RAM.

TABLE I
Results with benchmarks from T2

	#ins.	noMS	MS	MS+QR	MS+QR+TI	T2
Set1	449	212	220	228	238	245
Set2	472	245	252	262	276	279

TABLE II
Results with benchmarks from Jutge.org.

	#ins.	CppInv	T2
P11655	367	324	328
P12603	149	143	140
P12828	783	707	710
P16415	98	81	81
P24674	177	171	168
P33412	603	478	371

	#ins.	CppInv	T2
P40685	362	324	329
P45965	854	780	793
P70756	280	243	235
P81966	3642	2663	926
P82660	196	174	177
P84219	413	325	243

The first two considered sets of benchmarks are those provided by the T2 developers. Following the experiments in [8], we have set a 300 secs. timeout. To show the impact of the different techniques described in the paper, Table I presents the number of instances in each set (#ins.) and the number of those that we proved terminating with the following settings:

- *(noMS)* implements the generation of invariants and ranking functions using a translation to SMT(NA), but without using Max-SMT, i.e. with all constraints *hard*. The fact that this plain version can already prove many instances hints on the goodness of our underlying algorithm and the impact of using our NA-solver in this application.
- *(MS)* implements the generation of invariants and ranking functions using Max-SMT(NA), where the constraints imposed by the ranking function are added as *soft*.
- *(MS+QR)* adds to the previous case the possibility to use quasi-ranking functions.
- *(MS+QR+TI)* adds to the previous case the possibility to infer termination implications.

Note that every added improvement allows us to prove some more instances, while none is lost due to the additional complexity of the constraints generated.

Moreover, by looking into the results in more detail, we have observed that our tool and T2 complement each other to some extent: in Set1 CppInv can prove 7 instances which cannot be proved by T2, while we cannot prove 14 which can be handled by T2; similarly, in Set2 CppInv can prove 8 programs which cannot be proved by T2, while we cannot prove 11 that can be handled by T2. The average time in YES answers of T2 is 2.9 secs and of CppInv is 12.8 secs.

In Table II, we show the comparison of CppInv (with all described techniques) and T2 on our benchmarks from the programming learning environment Jutge.org, which is

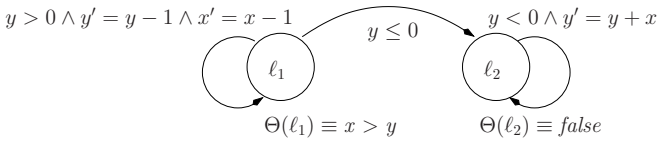


Fig. 4. Program that requires invariants from previous SCC's.

currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite consists of thousands of solutions written by students to 12 different programming problems. These programs can be considered challenging since most often they are not the most elegant solution but one with many more conditional statements than necessary (e.g., the largest instance we can successfully handle has nearly 700 transitions). Here, due to the size of the benchmark suites (see column #ins.), for the execution of both tools we have set a 120 secs. timeout. The average time in YES answers of T2 is 1.7 secs. and of CppInv is 1.6 secs. Note that, in order to run these benchmarks in T2, we have translated them into T2 format using our intermediate transition graph. This may be a disadvantage for T2, as it happens in the reverse way when CppInv is run on T2 benchmark set. In particular, we think the bad performance of T2 in sets P33412, P81966 and P84219 may be related to the way we handle division, which is crucial in these examples.

VI. Conclusions and Future Work

In short, the contributions of this paper are:

- a *novel Max-SMT constraint-based approach to proving termination*. Thanks to expressing the synthesis of a ranking function and a supporting invariant as a Max-SMT problem, we achieve a better guided and more fine-grained termination analysis than SMT-based methods. Max-SMT reveals to be a convenient framework for constraint-based termination analysis. In addition to our method, other techniques such as *unaffecting score maximization* [10] can be naturally modeled in Max-SMT.
- a *prototype of termination analyzer for (a subset of)* C++.

One of the shortcomings of our approach is that invariant synthesis is restricted to a single SCC. If invariants from previous SCC's have not been generated but are later required, our technique cannot prove termination. E.g., in the program shown in Fig. 4, the invariant $x > 0$ must be discovered at ℓ_1 so as to prove that the rightmost transition is finitely executable, although it is not necessary for proving that the leftmost loop is terminating. For future work we plan to develop techniques to overcome this kind of situations. A promising idea is to consider initiation conditions as soft: then the generated *quasi-invariants* represent what is missing from previous SCC's, and then can be propagated backwards. Alternatively, these quasi-invariants can be used to split the initial conditions of the current SCC. Finally, as a byproduct, this would allow us to solve the conditional termination problem as well.

Acknowledgment

This research was supported by Spanish MEC/MICINN under grant TIN 2010-21062-C02-01. We thank Jutge.org for providing benchmarks, and Byron Cook for giving us access to T2 and their benchmarks and for his helpful comments.

References

[1] D. Dams, R. Gerth, and O. Grumberg, "A heuristic for the automatic generation of ranking functions," in *Workshop on Advances in Verification*, 2000, pp. 1–8.
[2] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 67–81.
[3] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *VMCAI*, ser. LNCS, vol. 2937. Springer, 2004, pp. 239–251.
[4] A. Tiwari, "Termination of linear programs," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 70–82.
[5] M. Colón and H. Sipma, "Practical methods for proving program termination," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 442–454.
[6] A. Bradley, Z. Manna, and H. Sipma, "Linear ranking with reachability," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 491–504.
[7] J. Petit, O. Giménez, and S. Roura, "Jutge.org: an educational programming judge," in *SIGCSE*, ACM, 2012, pp. 445–450.
[8] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *CAV*, 2013, to appear.
[9] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear Invariant Generation Using Non-linear Constraint Solving," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 420–432.
[10] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013, pp. 47–61.
[11] A. Podelski and A. Rybalchenko, "Transition invariants," in *LICS*. IEEE Computer Society, 2004, pp. 32–41.
[12] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI*, ACM, 2006, pp. 415–426.
[13] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 81–95.
[14] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger, "Loopfrog: A Static Analyzer for ANSI-C Programs," in *ASE*, IEEE, 2009, pp. 668–670.
[15] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, "Proving conditional termination," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 328–340.
[16] P. Ganty and S. Genaim, "Proving termination starting from the end," in *CAV*, 2013, to appear.
[17] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.
[18] R. Nieuwenhuis and A. Oliveras, "On SAT Modulo Theories and Optimization Problems," in *SAT*, ser. LNCS, vol. 4121. Springer, 2006, pp. 156–169.
[19] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *ESOP*, ser. LNCS, vol. 3444. Springer, 2005, pp. 5–20.
[20] B. Cook, A. Podelski, and A. Rybalchenko, "Summarization for termination: no return!" *Formal Methods in System Design*, vol. 35, no. 3, pp. 369–387, 2009.
[21] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT Modulo Linear Arithmetic for Solving Polynomial Constraints," *J. Autom. Reasoning*, vol. 48, no. 1, pp. 107–131, 2012.
[22] J. A. Robinson and A. Voronkov, Eds., *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
[23] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 294–298.
[24] A. Podelski and A. Rybalchenko, "ARMC: the logical choice for software model checking with abstraction refinement," in *PADL*, ser. LNCS, vol. 4354. Springer, 2007, pp. 245–259.
[25] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl, "Automated termination analysis of java bytecode by term rewriting," in *RTA* Volume 6 of LIPIcs., Schloss Dagstuhl, 2010, 259–276.

On the Concept of Variable Roles and its Use in Software Analysis

Yulia Demyanova, Helmut Veith, Florian Zuleger
Vienna University of Technology

***Abstract*—Human written source code in imperative programming languages exhibits typical patterns for variable use, such as flags, loop iterators, counters, indices, bitvectors, etc. Although it is widely understood by practitioners that these patterns are important for automated software analysis tools, they are not systematically studied by the formal methods community, and not well documented in the research literature. In this paper, we introduce the notion of variable roles on the example of basic types (int, float, char) in C. We propose a classification of the variables in a program by variable roles which formalises the typical usage patterns of variables. We show that classical data flow analysis lends itself naturally both as a specification formalism and an analysis paradigm for this classification problem. We demonstrate the practical applicability of our method by predicting membership of source files to the different categories of the software verification competition SVCOMP 2013.**

I. Introduction

Programs written in imperative programming languages, such as C, Java, Perl, Python, share typical patterns of variable use, like flags, loop iterators, counters, indices, bitvectors, temporary variables, and so on. Experienced programmers have informal knowledge of these patterns, to which we refer as *variable roles*. For example, from the piece of C code `while(i<n) a[i++]=0;`, it can be deduced that `i` is a loop iterator and an array index. Similarly, from the statement `x&=y`, we can infer that `x` is a bitvector.

In common programming languages, there is no direct mapping from data types to roles - multiple roles can be associated with the same type. For example, in C, the type `int` can be used to store such different values as boolean, file descriptor, bitvector, and character literal. Moreover, it is not clear how to extend standard type systems for languages like C to express roles like array index, counter, and loop iterator. Additionally, one variable can have several roles simultaneously, like the variable `i` in the loop example above. In type systems, in contrast, one variable must be assigned one and only one type. Therefore, roles can not be considered simply as refined types.

Information about variable roles is implicitly contained in the structure of the source code, thus the roles can often be inferred by syntactic analysis. This can be done by analysing the expressions or statements of a given kind, for example, matching array indices in array subscripts. Alternatively, roles can be inferred by searching for code patterns, for example, `t=x; x=y; y=t;` is a typical pattern for a temporary variable `t`.

The notion of a variable role has two dimensions. In general, variable roles represent heuristics, which means that they can be systematically studied and analysed, but they need to be treated as auxiliary heuristic information. Thus, variable roles can *guide* a verification tool, but the *soundness* of a formal analysis must not depend on variable roles. Certain variable roles, however, provide *sound* information, which can be *relied upon* during verification, and thus these roles can be viewed as types. We will explore these two dimensions of variable roles in future work.

In this paper we define 14 variable roles with a standard data-flow analysis. Our definition serves at the same time as an algorithm to compute the roles. In order to choose the roles, we have manually investigated 5.2 KLOC of C code from the cBench benchmark [1]. We assigned roles to the variables of basic types such as int, float and char. When choosing the roles, we were inspired by typical programming patterns for variable use in real life programs. We have chosen the roles in such a way that a small number of roles is able to classify each occurring program variable in the programs we considered. We have implemented a prototype of a tool which maps basic-type variables in C programs to sets of roles.

As this short paper is reporting work in progress, we are currently exploring applications for variable roles. An important natural application is the use of variable roles to create abstractions in software verification or choose abstract domains through a better understanding of the program. For example, in C programs integer variables are used to store boolean flags, because there is no boolean type. When creating an abstraction for a C program, we know that the predicate `x==0` provides sufficient information about a boolean variable `x`. However, most state-of-the-art verification papers consider a program as a logical formula and either ignore such implicit information, or treat it as undocumented heuristics. For example, the ASTREE static analyser [2] relies heavily on human insight for selecting the right domain. Variable roles could be used for automating this process, e.g., to suggest the use of octagon or polyhedra domains for variables which occur only in linear operations, BDDs for boolean variables, etc. This will save a verification tool from enumerating all possible domains. After submission of this paper, we learnt about current work in this direction by the developers of the CPAchecker verification tool [3].

Another important application area of our method is to classify source files, for example, from benchmarks for different verification competitions, according to the relative number of occurrences of variable roles in them. To demonstrate

ISBN 978-0-9835678-3-7/13.

```
int x, y, n =1 0;
...
y =2 x;
while (x)
{
    n++3;
    x =4 x &5 (x-1);
}
```
a) bitvector, counter, iterator

```
int fd =1 open(path, flags);
int c, val =2 0;

while (read3(fd, &c, 1) > 0
    && isdigit4(c))
{
    val =5 10*val + c-'0';
}
```
b) character, file descriptor, linear

Fig. 1: Different patterns of use of integer variables

this point, we performed the following experiment with the benchmarks from the software competition SVCOMP 2013 [4]. The competition distinguishes several categories of source files, such as device drivers, embedded systems, concurrent programs, and so on. The classification is done by human experts, who manually analysed and comprehended the source code. With our tool we computed the frequency of different roles in each category and used this data to train a machine learning tool to predict the competition categories for new files. In a number of experiments, we randomly selected a subset of the competition source files for training and used the remaining source files to check our prediction against the human classification. Importantly, we used a machine learning technique not to infer roles, but rather to validate their predictive power. The results of the experiments are encouraging - the prediction is successful in more than 80% of the cases. We highlight that our choice of the roles was based on examples from cBench rather than SVCOMP.

The results of our experiment suggest that variable roles can be used to interpret experimental results for current verification tools. The strengths and weaknesses of the tools can be identified by computing code metrics in terms of the relative frequencies of the roles. This idea can be further elaborated for building a portfolio-solver where we first analyse the variable roles of the program under scrutiny and then select the verification tool that is best suited.

Contributions:

- We identify 14 variable roles that commonly occur in practical programs.
- We implement a prototype of a tool which assigns a set of roles to every basic-type variable in a C program.
- Using our tool and a machine learning technique, we predict the membership of a C program to a category of the SW verification competition SVCOMP 2013. We get encouraging results in a number of experiments.

II. Formalisation of variable roles

A. Examples

We will use the C programs of Figure 1 to informally introduce variable roles, whose formal definitions are given later in the section. In the programs we have assigned labels to the statements and expressions to which we refer from the text.

The program in Figure 1a calculates the number of non-zero bits of the variable `x`. In every loop iteration, a non-zero bit of `x` is set to zero and the counter `n` is incremented. The loop continues until all bits are set to zero. Although the variables `x` and `n` are declared of the same type `int`, they are used differently. For a human reading the program, the statements `n=0` and `n++` in the loop body signal that `n` is a counter. Indeed, `n` is used to count the number of loop iterations. On the other hand, the value of the variable `x` as an integer is not used in calculations, but rather individual bits in its binary representation matter.

We define the roles by restricting the operations in which a variable occurs. We require that a bitvector occurs in at least one bitwise operation (bitwise AND, OR or XOR), like the variable `x` in expression 5. We require that a counter variable only changes its value in an increment or decrement statement or gets assigned zero. The variable `n`, which is assigned in statements 1 and 3, satisfies these constrains.

The program in Figure 1b reads a decimal number from a text file and stores its numeric representation in the variable `val`. In contrast, the variables `fd` and `c` are used to store the output of the library functions `open()` and `read()` respectively. The difference between the two variables is that `c` is later used in calculations, while `fd` is only passed to the function `read()` as a black box because its value does not directly affect the result of the computations. One can conjecture that `c` is a character, because it is passed as an input to the function `isdigit()`, which checks whether its parameter is a decimal digit character. Even though `isdigit()` is declared to take a parameter of type `int`, the documentation states that the parameter is a character to be tested, cast to int [5].

We define character, file descriptor and linear roles as follows. We require that a character variable is assigned at least once a character literal (e.g., `c='a'`) or another character variable, or is used in a standard C function for manipulating characters (e.g., `c=getchar()` or `isdigit(c)`). A file descriptor is required to be used in a standard C function for manipulating files (e.g., `fd=open(path,flags)` or `read(fd,&c,1)`). A linear variable can be assigned only linear combinations of linear variables. The variables `c` and `val`, assigned in statements 3 and 2,5 respectively, satisfy the latter constraint.

B. Definition of the analysis

We define variable roles using classical intraprocedural dataflow analysis [6]. In this section we use the notation as follows. **Var** denotes the set of program variables, and **Num** denotes all scalar constant literals (e.g., 0, 0.5, 'a'). **S**, **E** and **B** denote the set of program statements, arithmetic and boolean expressions respectively. For the elements of these sets we use the same names in the lowercase version (e.g., var for a program variable).

For a program s$\in$ **S** the result of analysis R is computed using the function Res^R, which is defined as follows:

$$Res^R = Init^R \bigsqcup gen^R(s),$$

BITVECTOR Init = $\emptyset$, $\bigsqcup$ = $\cup$, c = **o**

$$\text{gen(var := e)} = \begin{cases} \{\text{var}\} & \text{if e ::= } e_1 \text{ bitop } e_2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{gen(if b then } s_1 \text{ else } s_2\text{)} = \text{gen(b)} \cup \text{gen}(s_1) \cup \text{gen}(s_2)$$

$$\text{gen}(s_1; s_2) = \text{gen}(s_1) \cup \text{gen}(s_2)$$

$$\text{gen(skip)} = \emptyset$$

$$\text{gen(while b do s)} = \text{gen(b)} \cup \text{gen(s)}$$

$$\text{gen(var) = gen(num)} = \emptyset$$

$$\text{gen}(e_1 \text{ bitop } e_2) = \text{IsVar}(e_1) \cup \text{IsVar}(e_2) \cup \text{gen}(e_1) \cup \text{gen}(e_2)$$

$$\text{gen}(e_1 \text{ arithop } e_2) = \text{gen}(e_1) \cup \text{gen}(e_2)$$

$$\text{gen(bitnot e)} = \text{IsVar(e)} \cup \text{gen(e)}$$

$$\text{IsVar(e)} = \begin{cases} \{\text{var}\} & \text{if e ::= var} \\ \emptyset & \text{otherwise} \end{cases}$$

LINEAR Init = **Var**, $\bigsqcup$ = $\setminus$, c = **f**

$$\text{gen(var:=e)} = \begin{cases} \{\text{var}\} & \text{if lin(e)=false} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{gen(if b then } s_1 \text{ else } s_2\text{)} = \text{gen}(s_1) \cup \text{gen}(s_2)$$

$$\text{gen}(s_1; s_2) = \text{gen}(s_1) \cup \text{gen}(s_2)$$

$$\text{gen(skip)} = \emptyset$$

$$\text{gen(while b do s)} = \text{gen(s)}$$

$$\text{gen(e)} = \emptyset$$

$$\text{lin(num)} = \text{true}$$

$$\text{lin(var)} = \begin{cases} \text{true} & \text{if var} \in \text{Res}^{LINEAR} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{lin}(e_1 + e_2) = \text{lin}(e_1) \wedge \text{lin}(e_2)$$

$$\text{lin}(e_1 * e_2) = \begin{cases} \text{lin}(e_2) & \text{if } e_1 \in \textbf{Num} \\ \text{lin}(e_1) & \text{if } e_2 \in \textbf{Num} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{lin}(e_1 \text{ bitop } e_2) = \text{lin(bitnot e)} = \text{lin}(e_1/e_2) = \text{false}$$

Fig. 2: Formal definition of roles BITVECTOR and LINEAR

where $Init^R \in \mathcal{P}(\textbf{Var})$ is the *initial* set of variables, the function $gen^R : \textbf{S} \cup \textbf{E} \cup \textbf{B} \rightarrow \mathcal{P}(\textbf{Var})$ maps every statement and expression to a set of *generated* variables, and the sign $\bigsqcup$ is used as a placeholder for a set operation and must be instantiated for each analysis.

Analysis R is therefore defined by a tuple ($Init^R$, $\bigsqcup$, gen^R, c), where c$\in \{\textbf{f}, \textbf{o}\}$ indicates how to evaluate Res^R. When c is defined as **f**, a fixed point of Res^R is computed, i.e. Res^R is iteratively recalculated until it does not change. When c is defined as **o**, Res^R is calculated in one iteration.

C. Example of role definition

In Figure 2 we formally define the analysis for the roles BITVECTOR and LINEAR. Due to the lack of space we give only an informal definition of the remaining roles in Table I. We now show step by step the computation of the BITVECTOR and LINEAR roles for the example program in Figure 1a.

The analysis for the role BITVECTOR starts with the empty set ($Init = \emptyset$). The operation $\bigsqcup$ is defined to be set union, and the result set is calculated in one iteration (c = **o**). When statement 4 is processed, the variable `x` is added to the result set because in this statement `x` is assigned the result of a bitwise AND operation. At expression 5, the variable `x` is also added to the result set because `x` occurs in a bitwise operation. After that, the result set does not change anymore, and the analysis yields the result {`x`}, as shown in Figure 3a.

TABLE I: Informal definition of variable roles

Role name	Informal definition
SYNT_CONST	not assigned any value in the program
CONST_ASSIGN	assigned only numeric literals or CONST_ASSIGN variables
COUNTER	assigned only in increment and decrement statements, or assigned zero
LINEAR	assigned only linear combinations of LINEAR variables
BOOL	assigned only 0,1, the result of a boolean expression or BOOL variables
INPUT	modified by an external function
BRANCH_COND	occurs in the condition of an if statement
BITVECTOR	occurs in a bitwise operation or assigned the result of a bitwise operation
UNRES_ASSIGN	assigned a pointer dereference or modified by a function
CHAR	assigned only character literals, CHAR variables, or passed to a library function which manipulates characters
LOOP _ITERATOR	occurs in the condition of a loop and assigned in the loop body
FILE_DESCR	passed to a library function which manipulates files
ARRAY_INDEX	occurs in an array subscript expression
ARRAY_SIZE	passed to a memory allocating library function

BITVECTOR	
label	gen(s)
4,5	{x}
Init(vd)=$\emptyset$, Res={x}	

a) bitvectors

LINEAR		
Iter.	label	gen(s)
1	4	{x}
2	2	{y}
Init(vd)={x,y,n}, Res={n}		

b) linear variables

Fig. 3: Step-by-step computation of roles

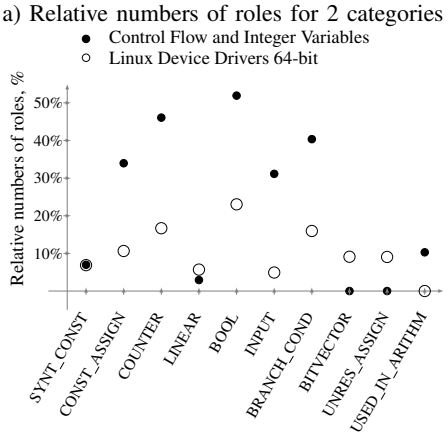
The analysis for the role LINEAR is computed as a fixed point of the function Res^R (c = **f**). It starts with the set **Var** of all program variables, which evaluates to {`x,y,n`}. The operation $\bigsqcup$ is defined to be set subtraction. In the first iteration, the variable `x` is excluded from the result set at statement 4 because it is assigned a non-linear expression. In the second iteration, the variable `y` is excluded from the result set at statement 2 because it is assigned `x`, which does not belong to the result set. In the third iteration, the result set does not change, and the analysis yields the result {`n`}, as shown in Figure 3b.

III. Implementation and experiments

We used the clang compiler [7] to implement a prototype of a tool, which assigns a subset of variable roles to every basic-type variable. The current implementation does intraprocedural analysis and does not include a pointer analysis. We replace all function calls (e.g., `c=getchar()`) and pointer dereferences (e.g., `n=*ptr`, `n=arr[i]`) with fresh variables. For example, the statement `c=getchar()` would be rewritten as `c=`t_1, and in the LINEAR analysis the variable t_1 would not be excluded from the result set, but rather assigned the role "unresolved assignment", which is a trade-off between soundness and precision.

We ran two experiments. In the first one we computed the relative number of the occurrences of each role in every category. We calculated it by summing up the numbers in all files of a category and normalising them by the total number of variables in these files. The results for the categories "Control

a) Relative numbers of roles for 2 categories



b) Prediction error in different settings

Training set, % of all files	Error of choice 1	Error of choice 2
90%	15.94%	2.90%
80%	14.81%	5.93%
70%	16.20%	7.98%
60%	19.77%	7.98%
50%	18.60%	8.54%

Fig. 4: Comparison of categories and automatic classification of files

Flow and Integer Variables" and "Linux Device Drivers" are shown in Figure 4a. We observed higher frequencies of boolean flags and branching operations, counters, arithmetic operations and constant assignments in the first category, and high numbers (in comparison to other categories in SVCOMP) of bitvectors and pointers in the second one.

In our main experiment, whose summary was given in Section I, we used machine learning to create a classifier for source files into the categories of the competition SVCOMP as a function of the frequencies of variable roles in a file. Since a program would typically share similarities with several categories, we used a multiclass vector support machine [8] to predict the category of a source file with some probability. For example, we predict that a file is a driver with the probability 60%, a concurrent program with the probability 35%, and so on. We translated the relative numbers of roles into the input format of the machine learning tool Weka [9] as follows: each source file represented one training example with the category corresponding to the class, and relative numbers of roles representing the vector of float attributes. We ran the experiments for varying sizes of the training sets from 90% to 50% of all files and analysed the remaining files by our tool. Figure 4b shows the percentage of the files for which the most likely category (second column) or the two most likely categories (third column) do not include the actual file category.

IV. Related work

The term variable roles was inspired by the work in program comprehension [10] which informally defines roles as *patterns of how variables are initialised and updated*. The authors have defined nine roles, implemented a tool for assigning roles to variables using static analysis, and evaluated it on Pascal programs from textbooks. The work leaves open the question of formalising the notion of variable roles as well as of the possibility of applying the method to real-word programs.

The commercial bug finding tool Coverity [11] uses implicit knowledge in the form of programmer's *beliefs*, i.e. propositional statements about program variables and functions. The authors use static analysis to extract two types of statements – the sound statements which follow from the requirements of safety, non-redundancy and reachability of the code (e.g., "a pointer is not null") and hypotheses which follow from the statistics of observations (e.g., "calls to functions *f()* and *g()* should be paired").

Rondon et al. [12] use predicate abstraction over a fixed set of predicates to infer so called *liquid* types, i.e. types refined with a conjunction of propositional predicates (e.g., $x{>}0 \wedge x{<}5$). We consider this approach to be complementary to ours, because it does not use any information from the source code other than the transition relation, and concentrates on arithmetic properties of variables.

Variable names and comments as an additional source of knowledge about a program have been systematically studied in program comprehension. The *Latent Semantic Indexing* technique [13] allows to query the program source code using words in natural language, based on the number of occurrences of the words in variable names and comments. A study has been made of the naming rules for variables in real-word programs [14], and of expanding abbreviated identifiers to full words [15]. We plan to use these techniques in future work.

Acknowledgment

This work is supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED and ICT12-059.

References

[1] http://ctuning.org/wiki/index.php/CTools:CBench.
[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astrée analyzer," in *Programming Languages and Systems*. Springer, 2005, pp. 21–30.
[3] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein, "Domain types: Selecting abstractions based on variable usage," CoRR, Tech. Rep. abs/1305.6640, 2013.
[4] http://sv-comp.sosy-lab.org/2013/benchmarks.php.
[5] http://www.gnu.org/software/libc/manual/pdf/libc.pdf.
[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer-Verlag New York Incorporated, 1999.
[7] http://clang.llvm.org.
[8] J. Weston and C. Watkins, "Multi-class support vector machines," Department of Computer Science, Royal Holloway, University of London, Tech. Rep. CSD-TR-98-04, 1998.
[9] http://www.cs.waikato.ac.nz/ml/weka.
[10] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs," in *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments*, 2002, pp. 37–39.
[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.
[12] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2008, pp. 159–169.
[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.
[14] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.
[15] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 213–222.

Author Index

FMCAD 2013 SPONSORS