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 twelfth conference in the series, FMCAD 2012, was held in Cambridge, United Kingdom, 22–25 October, at Microsoft's Cambridge research laboratory.

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 yearly. It provides a leading international forum for researchers and practitioners in academia and industry to present and discuss novel methods, technologies, theoretical results and tools for formal reasoning about computing systems.

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. The Hardware Model Checking Competition (HWMCC) was co-located with the conference this year. The FMCAD 2012 conference received 71 submissions (after discounting withdrawn submissions).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, 25 submissions were eventually selected for presentation at the conference, 21 as regular papers and 4 as short papers. The accepted papers covered topics ranging from model checking and solver technology to design for verification, synthesis, debug and testing. Moreover, they addressed a broad spectrum of abstraction levels ranging and a wide variety of topics including the verification of multi-threaded programs and automatic lock insertion, analog and mixed-signal systems, efficient models for large memories, symbolic synthesis of small circuits, automated debugging of missing input constraints in formal verification systems, software model checking, floating-point logic with semantic abstraction, symbolic trajectory evaluation, reachability analysis, and bounded model checking.

Besides reviewed submissions, our program was enriched by four invited tutorial speakers. *Jasmin Fischer*, a researcher at Microsoft's Cambridge research laboratory, talked about "Formal Methods in Cell Biology", which deals with the application of formal techniques to systems biology. *Torsten Schaub* from the University of Potsdam, Germany, gave a talk about "Answer Set Programming". *Eric Feron*, from Georgia Tech, presented "Formal Methods for Aerospace Applications". *Alessandro Cimatti*, from Fondazione Bruno Kessler, gave a presentation called "Application of SMT Solvers to Hybrid System Verification". The Keynote speaker for the conference, *Tony Hoare*, from Microsoft Research Cambridge, gave a talk called "Algebra of Concurrent Design". There was also an invited talk from ARM by *Daryl Stewart* called "Formal for Everyone - Challenges in Achievable Multicore Design and Verification". *Maher Mneimneh* from Atrenta Inc. chaired a panel session called "Model Checking in the Cloud."

The 2012 Proceedings of FMCAD are available through the ACM Digital Library, at IEEE Xplore Digital Library, or as a free download from the FMCAD website.

We would like to sincerely thank our industrial sponsors for their financial support of FMCAD 2012: ARM, Atrenta Inc, Centaur Technology, IBM Corp., Intel Corp., Jasper Design Automation, Mentor Graphics, Microsoft Research Cambridge, NEC Laboratories America, OneSpin Solutions. We would also like to acknowledge the continuous support of FMCAD Inc. We owe a large debt to this year's organizing committee, composed of *Samin Ishtiaq* (Local Arrangements), *Stefano Quer* (Publication), *Slava Bulach* and *Fahim Rahim* (Publicity), *Rolf Drechsler* (Tutorials), as well as to *Maher Mneimneh* for organizing the panel session. We would also like to thank the members of the FMCAD Steering Committee: *Jason Baumgartner*, *Aarti Gupta*, *Warren Hunt*, *Panagiotis Manolios*, and *Mary Sheeran* – for their kind advice during the conference preparation process. A big thanks goes 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. We also owe a large debut to *Cara Freeman* at Microsoft for local arrangements support and to Microsoft Research Cambridge for kindly hosting the conference at their laboratory. Last, but not least, the conference would not be possible without all the authors who submitted papers and all the attendees.

Gianpiero Cabodi and *Satnam Singh* (chairs)

Conference Organization

Program Co-Chairs

Gianpiero Cabodi, Politecnico di Torino, Italy
Satnam Singh, Google, USA

Local Arrangement Chair

Samin Ishtiaq, Microsoft Research Cambridge, UK

Publication Chair

Stefano Quer, Politecnico di Torino, Italy

Publicity Chairs

Slava Bulach, Robert Bosh GmbH, Germany
Fahim Rahim, Atrenta Grenoble, France

Tutorial Chair

Rolf Drechsler, University of Bremen, Germany

Steering Committee

Jason Baumgartner, IBM, USA
Aarti Gupta, NEC Labs America, USA
Warren Hunt, University of Texas at Austin, USA
Panagiotis Manolios, Northeastern University, USA
Mary Sheeran, Chalmers University of Technology, Sweden

Program Committee

Jason Baumgartner, IBM
Armin Biere, Johannes Kepler University
Per Bjesse, Synopsys
Roderick Bloem, Graz University of Technology
Gianpiero Cabodi, Politecnico di Torino
Alessandro Cimatti, Fondazione Bruno Kessler
Byron Cook, Microsoft Research
Bruno Dutertre, SRI international
Steven German, IBM
Mark Greenstreet, University of British Columbia
Aarti Gupta, NEC Labs America
Youssef Hamadi, Microsoft Research
Alan Hu, University of British Columbia
Warren Hunt, University of Texas at Austin
Barbara Jobstmann, Verimag/CNRS
Kevin Jones, City University London
Daniel Kroening, Computer Science Department, Oxford University
Thomas Kropf, Robert Bosch GmbH and University of Tuebingen
Panagiotis Manolios, Northeastern University
Joao Marques-Silva, University College Dublin
Arie Matsliah, CWI Amsterdam
Ken McMillan, Microsoft Research
Tom Melham, Oxford University

External Reviewers

Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)

Formal Methods in Cell Biology

Jasmin Fisher

Microsoft Research, Cambridge, United Kingdom

Abstract

Biological systems are extremely complex reactive systems. They operate as highly concurrent programs with millions of entities running in parallel and communicating with each other under various environmental conditions. Understanding how living systems operate in such harmony and precision, and how this harmony is being disrupted in diseased states, are key questions in biological and medical research. Due to their enormous complexity, the comprehension and analysis of living systems is a major challenge. Over the last decade various efforts to tackle this problem concentrate on a new approach called Executable Biology focused on the construction and analysis of executable models describing biological phenomena. Over the years, these efforts have demonstrated successfully how the use of formal methods can be beneficial for gaining new biological insights and even directing new experimental avenues. In this tutorial, I will survey some of the major efforts in this direction, using formal verification, synthesis and the design of new tools to reason about information processing during cells decision-making, organisms development, and molecular mechanisms underlying various human cancers.

Answer Set Programming

Torsten Schaub[1]

University of Potsdam, Germany
Email: torsten@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP; [1], [2], [3], [4]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capacities. ASP is particularly suited for modeling problems in the area of Knowledge Representation and Reasoning involving incomplete, inconsistent, and changing information. From a formal perspective, ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT). Applications of ASP include automatic synthesis of multiprocessor systems, decision support systems for NASA shuttle controllers, Linux package configuration, reasoning tools in systems biology, and many more. The versatility of ASP is also reflected by the ASP solver `clasp` [5], [6], [7], developed at the University of Potsdam, winning first places at first places at ASP, CASC, MISC, PB, and SAT competitions. This short tutorial presents a practical introduction to ASP, aiming at using ASP languages and systems for solving application problems. Starting from the essential formal foundations, it introduces ASP's solving technology, modeling language and methodology, while practically illustrating the overall solving process by examples.

References

[1] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proceedings of ICLP'88, The MIT Press (1988) 1070–1080

[2] Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273

[3] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)

[4] Gelfond, M.: Answer sets. In Lifschitz, V., van Hermelen, F., Porter, B., eds.: Handbook of Knowledge Representation. Elsevier (2008) 285–316

[5] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proceedings of IJCAI'07, AAAI Press/The MIT Press (2007) 386–392

[6] Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver clasp: Progress report. Proceedings of LPNMR'09. Springer (2009) 509–514

[7] Potassco, the Potsdam Answer Set Solving Collection. `http://potassco.sourceforge.net/`

[1]Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

Formal methods for Aerospace Applications

Eric Feron

Georgia Tech, USA

Abstract

Formal methods are being progressively incorporated in the aircraft and spacecraft software design and verification process and become commonplace elements of the aerospace industry. Five aerospace software system experts will present their views on this process and where it is headed.

Focusing first on design issues, PETE MANOLIOS (Northeastern University, USA) will discuss design aspects and costs of commercial air transport vehicles, including integrated modular avionics, verification costs, and system integration. He will then discuss how new verification technology is used to algorithmically synthesize an optimal architecture subject to high level constraints. This work will be illustrated by a case study involving the Boeing 787 Dreamliner.

MARC PANTEL (IRIT, France) will then discuss safety requirements as a key aspect of the development of embedded systems in avionics. He will discuss the current regulations linking safety requirements to software design guidelines. He will then discuss novel approaches to model driven software development, using formal models and verification activities at the various steps of the development cycle. Experiments conducted in relation with European avionics companies will be described.

Moving then towards analysis methods, GUILLAUME BRAT (NASA, USA) will discuss sound, complete, precise, and scalable static analysis of flight control systems. He will introduce the IKOS static analysis framework, whose intellectual foundation is abstract interpretation. He will insist on compositional verification, a necessary tool for to make formal methods scale up to real, avionics systems. He will address the component-based development approach of these systems.

ERIC FERON (Georgia Tech, USA), and PIERRE-LOIC GAROCHE will discuss the application of the methods introduced above to control software, a narrow, but essential component of any safety-critical software system. They will then describe a possible evolution of the current development process of aircraft control systems towards more formalism (through a combination of formal proof and proof replay). They will discuss the static analysis of the behavior of the controller (stability and other non linear properties), and the static analysis of the safety architecture of the controller.

Application of SMT Solvers to Hybrid System Verification

Alessandro Cimatti

Fondazione Bruno Kessler, Trento, Italy

Abstract

Hybrid automata are a widely used framework to model complex critical systems, where continuous physical dynamics are combined with discrete transitions. Application areas include automotive, railway, aerospace, and industrial production.

The expressive power of Satisfiability Modulo Theories (SMT) solvers can be used to symbolically model networks of hybrid automata, using formulas in the theory of reals.

In this tutorial, we survey state-of-the-art SMT-based verification for hybrid systems.

We show how SAT-based techniques such as bounded model checking, k-induction, predicate abstraction, and IC3, can be naturally lifted to the SMT case. The expressive power of the SMT framework allows us to exploit a local time semantics, where the timescales of the automata in the network are synchronized upon shared events. The approach fully leverages the advanced features of modern SMT solvers, such as incrementality, unsatisfiable core extraction, and interpolation.

We then concentrate on the problem of scenario-based verification, i.e. checking if a network of hybrid automata accepts some desired interactions among the components, expressed as Message Sequence Charts (MSCs).

We conclude by investigating the problem of requirements analysis for hybrid systems.

Algebra of Concurrent Design

Tony Hoare

Microsoft Research, Cambridge, United Kingdom

Abstract

I introduce some familiar algebraic laws governing the operators of sequential and concurrent composition of designs. They can be combined with the familiar operators of propositional calculus. The resulting logic seems to apply equally to hardware design and to software design; and perhaps also to the planning of other designs and plans for behaviour that evolves in space and time.

Efficient Predictive Analysis for Detecting Nondeterminism in Multi-Threaded Programs

Arnab Sinha, Sharad Malik
Princeton University
{*sinha,sharad*}*@princeton.edu*

Aarti Gupta
NEC Laboratories America
agupta@nec-labs.com

***Abstract*—*Determinism* is often a desired property in multi-threaded programs. A multi-threaded program is said to be deterministic if for a given input, different thread interleavings result in the same system state in the execution of the program. This, in turn, requires that different interleavings preserve the values read by each read operation. A related, but less strict condition is for the program to be *race-free*. A deterministic program is race-free but the converse may not be true. There is much work done in the *static analysis* of programs to detect races and nondeterminism. However, this can be expensive and may not complete for large programs in reasonable time. In contrast to static analysis, *predictive analysis* techniques take a given program trace and explore other possible interleavings that may violate a given property – in this case the property of interest is determinism. Predictive analysis can be sound, but is not complete as it is limited to a specific set of program runs. Nonetheless, it is of interest as it offers greater scalability than static analysis. This work presents a predictive analysis method for detecting nondeterminism in multi-threaded programs. Potential cases of nondeterminism are checked by constructing a causality graph from the thread events and confirming that it is acyclic. On average, the number of graphs analyzed per benchamrk is one per potential case of nondeterminism, thereby ensuring that it is efficient. We demonstrate its application on some benchmark Java and C/C++ programs.**

I. Introduction

Writing correct and efficient multi-threaded programs is widely accepted as a challenging task. The wide range of possible concurrency errors makes it inherently harder than writing sequential programs [15], [26], [28]. Given the same input, the different runs of a multi-threaded program may produce different outputs because the threads interleave in different ways. This makes it hard to replicate and debug errors through traditional testing methods. These errors are referred to as “Heisenbugs” [2]. The potential *nondeterminism* of multi-threaded programs lies at the core of these Heisenbugs. For this and other reasons, *determinism* is often a desired property in multi-threaded programs. A multi-threaded program is said to be deterministic if for a given input, different thread interleavings result in the same system state in the execution of the program. It is important to consider when the system state is observed. If it is observed only at the end of the program execution, then individual read events may not need to read the same value across different interleavings. However, if the system state is continuously observed, then each read event must read the same value in all possible interleavings. We consider this case. Further, for ease of analysis we consider the stricter condition that each read event reads the value from the *same* write event in all interleavings. This restriction is consistent with other work in predictive analysis [8], [34], and can be supplemented with program analysis to consider specific values rather than specific events, if desired.

A related but less strict condition is a *datarace*. A pair of shared memory accesses are said to be *conflicting* if they are performed by different threads and at least one of them is a write. Also, the events are *unsynchronized* if the threads do not use an explicit mechanism such as locks to prevent the accesses from being simultaneous. A datarace is defined as two conflicting and unsynchronized data accesses. *A deterministic program is race-free but the converse may not be true*. The following example in Fig. 1 illustrates this further.

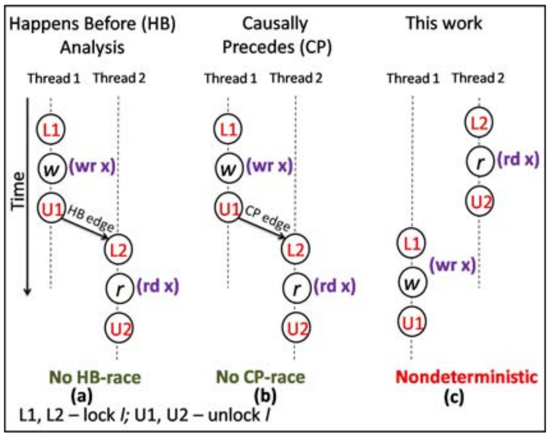


Figure 1. A deterministic program is race-free but the converse may not be true. (‘Causally precedes’ is defined in [35].)

Consider the example in Fig. 1. In this example, there is a pair of conflicting shared memory accesses, each under the lock-scope of the same lock variable l. Let events L1 and L2 be ‘*acquire lock*’ events on l. Similarly, let events U1 and U2 represent ‘*release lock*’ events on l. In (a), we show a standard Happens Before (HB) analysis for lock operations. The two events U1 and L2 are ordered by HB, as indicated by the (U1, L2) edge and hence there is no race. Next, in (b), we consider the *causally precedes* (CP) analysis proposed by Smaragdakis et al. [35]. Due to the presence of conflicting accesses (w and r) within the lock-scopes, U1 causally precedes L2 introducing the CP edge from U1 to L2. Hence, there is no *CP-race*. However, observe that in another interleaving (c), where the lock-scopes swap order, the following different order is possible in an interleaving: U2 happens before L1. Thus, while the program is race free, it is nondeterministic because the read event (rd x) reads from a different write event in the interleaving (c) compared to the interleaving in (a) and (b).

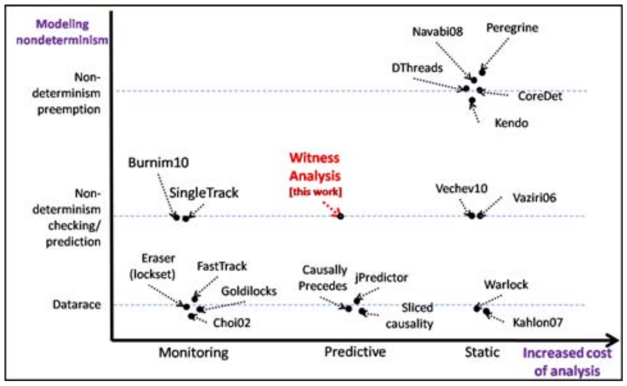


Figure 2. Classification of race and nondeterminism detection techniques based on cost of analysis: Burnim10 [6], SingleTrack [32], Eraser [33], FastTrack [13], GoldiLocks [11], Sliced Causality [7], jPredictor [8], Causally precedes [35], CoreDet [3], Kendo [29], DThreads [25], Navabi08 [27], Peregrine [10], Vaziri06 [37], Warlock [36], Kahlon07 [20], Choi02 [9], Vechev10 [38]

There is much work done in *static analysis* of programs to detect races and nondeterminism [36], [20], [9], [3], [29], [25], [27], [10], [37] as shown in Fig. 2. Among these, *deterministic multi-threading* (DMT) has attracted a lot of interest recently [25], [10]. DMT deterministically schedules the threads such that the values read by the read operations are preserved. The static analyses for detection or finding schedules can be expensive and may not complete for large programs within reasonable time.

The other end of the spectrum is *monitoring*-based solutions [33], [13], [11]. Although monitoring-based solutions are scalable and sound, the analysis is based only on the runs that are actually executed. In contrast, *predictive analysis* techniques take a given program trace and explore other possible interleavings that may violate a given property [35], [8], [7]. This helps to enhance coverage of a given test input to a larger set of thread interleavings. Predictive analysis can be sound but it is not complete as it may not cover the entire program.

In this work, we adopt a predictive analysis technique for detecting nondeterminism. This provides an effective trade-off between cost and coverage. Our technique is based on the partial order permitted by a trace combined with the reasoning for locks. This technique is fast because it searches a reduced set of sufficient interleavings. Potential cases of non-determinism are checked by constructing a causality graph from the thread events and confirming that this is acyclic. We demonstrate its application on some benchmark Java and C++ programs. Our results show that the average number of graphs analyzed per benchmark is one per potential case of nondeterminism.

This work makes the following contributions:

- It presents a sound and complete[1] predictive analysis technique for checking determinism of multi-threaded programs. It reports only feasible cases of nondeterminism and thus avoids false positives that would require additional test execution after the analysis.
- The proposed technique requires search over a reduced set of sufficient interleavings and hence is fast.
- The technique has been implemented and experimental results on C/C++ and Java benchmark programs are very promising.

II. Preliminaries

We consider a multi-threaded program consisting of a set of *threads* $T_1, T_2 \ldots, T_k$ and a set of *shared variables*. Let $\{1, \ldots, k\}$ be the set of thread indices. The remaining aspects of the program, including the control flow and the expression syntax, are intentionally left unspecified for generality.

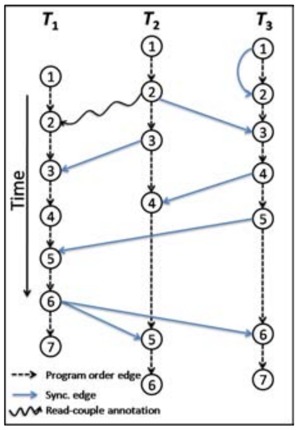


Figure 3. The partial order graph with vertices representing events and the dashed and solid edges are program order and sync. edges respectively. The read-couple annotations are indicated by the squiggly arrows.

Program Trace Model: An execution trace $\rho = e_1, e_2, \ldots e_n$ is a sequence of events, e_i, $i \in \{1, \ldots, n\}$, each of which is an instance of a *visible* operation during the execution of the program. The visible operations are: read/write accesses to shared variables and synchronization operations such as wait, notify, notifyall, lock acquire/release and thread fork/join. An event is represented as a 5-tuple $(tid, eid, type, var, child)$, where tid is the thread index ($tid \in \{1, \ldots, k\}$), eid is the event index (that starts from 1, and increases sequentially within a thread), $type$ is the event type, var is either a shared variable (in read/write operations) or a synchronization object, $child$ is the child thread index (in thread create/join). The event type is one of $\{read$, $write$, $fork$, $join$, $acquire$, $release$, $wait$, $notify$, $notifyall\}$.

An execution trace ρ is the observed interleaving of events across the threads and provides a total order on these events. We derive the required partial order for this trace by

[1]over all interleavings of events in the given trace, not over the entire program

retaining only the set of must-happen-before constraints as described below.

Partial Order Graph: Let $G(V, E)$ be a partial order graph such that $V(G)$ is the set of vertices, each of which represents an event in the trace (we use vertices and events interchangeably when the context is clear). Fig. 3 is an example partial order graph with three threads. The number inside each vertex is the eid within the thread. A directed edge (a, b) in $E(G)$ (the set of edges) is either a program order edge, or a synchronization (sync.) edge.[2] Program order edges are indicated by dotted arrows and sync. edges by solid arrows in Fig. 3. An edge in $E(G)$ is referred to as a *partial order edge*.

We note that locks are not added as sync. edges in $E(G)$. The mutual exclusion due to locks is considered separately by our analysis. We also give special consideration to write-read pairings. If event b reads the value written by event a, then the pair (a, b) is defined as a *read-couple*. A read-couple is indicated by a squiggly arrow annotation in the partial order graph G. Note that this is not included in the edge set $E(G)$. In a different interleaving τ, if b reads from a different event c, we say that the read-couple for b, and the read b in ρ is *broken* in τ.

Locked Scope: A locked scope, denoted as $[e_i \ldots e_j]_l$, is defined as the sequence of events $e_i \ldots e_j$ after an '`acquire lock` l' event and before a '`release lock` l' event, where l is a lock-variable. Note that the sequence of events $e_i \ldots e_j$ and lock acquire/release events belong to the same thread.

III. Predictive Analysis of Nondeterminism

We assume that the shared variables are implicitly written (or initialized) at the beginning of the execution. Similarly, they are all implicitly read at the end of the program execution. Given the same inputs, if a read instruction of a shared variable reads the value from the same write operation in all interleavings, it is referred to as a *view-preserving* read. Otherwise, the read is *non-view-preserving*. This is related to the well-known notion of view equivalence in database transactions [30].

Definition 1: **[Program Nondeterminism]** We define a multi-threaded program to be nondeterministic iff there exists at least one non-view-preserving read.

Writer, Readers and Challengers: In the given trace, there can be several read operations reading the value written by a single write operation, w. w is referred to as the *writer*. Any read event that reads the value written by w is denoted as *reader* of w. Let $R(w)$ be the set of readers of w. Any write operation c, other than w that writes the same shared variable is denoted as a *challenger* of w. It is named so since it challenges the set of read-couples induced by w (i.e. $\{(w, r)$ where, $r \in R(w)\}$) as in an alternate interleaving r may read from c instead of w, thus breaking the read couple (w, r). Let, $C(w)$ be the set of all challengers of writer w.

Problem Formulation: We aim to detect nondeterminism over alternate interleavings of events of a given trace ρ. Thus, we address the following problem: *given a trace ρ and a read-couple (w, r) in ρ, is there a challenger c such that it breaks (w, r) in another interleaving τ?*

For a pair of events e_1 and e_2 and an interleaving, let $e_1 \mapsto e_2$ represent "*e_1 precedes e_2 in the interleaving*". For a given triplet (w, r, c) and a partial order graph G, where $r \in R(w)$ and $c \in C(w)$, the read-couple is broken in an interleaving τ, when any of the following orders is present in τ: (1) $c \mapsto r \mapsto w$, or (2) $w \mapsto c \mapsto r$, or (3) $c \mapsto r$ and w does not occur in τ. In each case, r does not read from w in τ. We refer to these orders as *witnesses of nondeterminism* and the interleaving containing a witness as a *witness interleaving*. In cases (1) and (2), w, r and c are the events of the witness and in case (3), c and r are the events of the witness. A triplet is said to be nondeterministic if it can provide a witness of nondeterminism.

Central Idea: There are two phases in our analysis for each witness. For a certain witness ω to exist in an interleaving τ, τ must satisfy the orderings between the members of ω in addition to the HB constraints imposed by program-order, synchronization and possibly between locked scopes. Let $G'(\omega)$ be the graph after incorporating all the mentioned constraints to G in the form of ordering edges but not including any consideration of locked scopes. $G'(\omega)$ cannot contain a cycle since τ must be a total order of events satisfying the ordering constraints imposed by $G'(\omega)$. Thus, in the first phase of our analysis, we check for a cycle in $G'(\omega)$. Presence of cycle in $G'(\omega)$ entails the witness to be infeasible (*necessary condition* for feasibility of witness). (This phase is similar to a Universal Causality Graph (UCG)-based analysis [23]. We provide a detailed comparison later.) However, absence of a cycle in $G'(\omega)$ does not guarantee feasibility of witness. This is because we still need to consider the locked scopes. For each pair of mutually exclusive locked scopes LS_1 and LS_2, either LS_1 HB LS_2 or LS_2 HB LS_1. Since this holds for each pair of mutually exclusive locked scopes, we need to consider all possible combinations of such HB constraints. For d such pairs, there will be 2^d combinations. These choices need to be explored by augmenting G' with each of these 2^d combinations of HB constraints. In the second phase of our analysis, we construct all such possible graphs obtained by augmenting $G'(\omega)$. Let $G''(\omega)$ be one such graph. The witness is infeasible if and only if all 2^d $G''(\omega)$ graphs contain cycles (*sufficient condition* for feasibility of witness). We now describe these two phases in detail below.

[2]HB edges between $fork$ event in parent thread and first event in child thread, between $wait$ and $notify$ events, and between last event in child thread and $join$ event in parent thread are sync. edges.

A. Necessary Condition for Witness: Witness Order Graph

Let ω be a witness in an interleaving τ. We consider ordering constraints imposed by ω on τ. Note that G already contains program order and synchronization constraints that τ must obey. We now augment G to $G'(\omega)$ to include additional ordering constraints imposed by the witness ω. $G'(\omega)$ is referred to as the *witness order graph*. The orders imposed by ω are reflected by adding additional edges to $G'(\omega)$ denoted as *witness order edges*.

We construct $G'(\omega)$ specific to witness ω as follows. (Henceforth, we refer to $G'(\omega)$ as G' when ω is clear from the context.) Without loss of generality consider ω to be of the type $c \mapsto r \mapsto w$. We add witness order edges (c, r) and (r, w) to G'.

For each observed read-couple (a, b) in ρ besides (w, r) in ω, we add a read-couple edge (a, b) to $E(G')$. In addition, the *induced edges* [23] are added to $E(G')$ as described below. For a pair of locked scopes guarded by same lock-variable ($[u, \ldots, v]_l$ and $[x, \ldots, y]_l$, say), we add an induced edge (v, x) if there is path from u to y in G'[3]. However, if neither v precedes x and nor y precedes u in G' i.e. the locked-scopes are unordered, then the locked-scopes are said to have a *choice* between edges (v, x) and (y, u) in terms of the HB relation between them. This choice will be dealt with later. Fig. 4(a) shows a multi-threaded program trace where x and y are shared variables. The variable x is being written by events c and w in thread T_1 and read by event r in thread T_2 respectively. Event e_2 in thread T_1 assigns the address of x to variable y. Next, in thread T_2, the value of y is read in a local variable b. The events e_5, r and e_6 execute in thread T_2, if b is non-null. The partial order graph G in Fig. 4(b) corresponds to the multi-threaded program trace in Fig. 4(a). Further, Fig. 4(c) shows the witness order graph for the same program trace and witness $c \mapsto r \mapsto w$. The edge (e_3, e_5) is induced by (e_2, e_4) and the presence of locked scopes $[e_1, \ldots, e_3]_l$ and $[e_5, \ldots, e_6]_l$. Note that insertion of one induced edge can trigger insertion of another induced edge if the locked-scopes are nested or overlapping.

G' now contains the following four kinds of ordering constraints due to G (program order edges + sync. edges), witness order edges (including locked scope analysis), read-couple edges except (w, r), and the induced edges due to mutual exclusion of locked scopes. Locked scope analysis enforces the mutual exclusion constraint. However, when combined with the ordering enforced by a specific witness, the mutual exclusion constraint can lead to an ordering constraint which can be added to the partial order ordering constraints [23].

In Fig. 4(c), G' has a cycle ($r \to w \to e_1 \to e_2 \to e_4 \to e_5 \to r$). Since this cycle represents orderings corresponding to the edges in G', at least one of these orders is not possible.

[3]Presence of a path from u to y in G' implies that $[u, \ldots, v]_l$ must be entirely executed before starting the execution of $[x, \ldots, y]_l$.

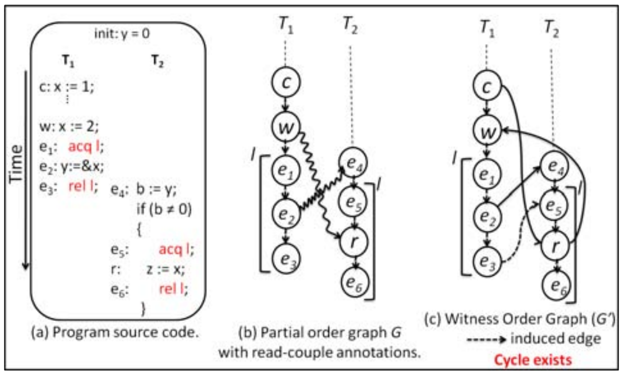


Figure 4. The partial order graph G and the witness order graph $G'(\omega)$, where ω is $(c \mapsto r \mapsto w)$ for the example program source code in (a).

Specifically, in this case, the (e_2, e_4) read-couple will be broken in τ. Therefore, the read for e_4 in τ may result in a different value from the read in the original trace ρ. This may alter the program flow so that the event r may not even happen in τ. In this case the witness is said to be infeasible as τ may not contain r.

Let (w', r') be a read-couple in ρ that is broken in τ. Let x be an event in witness ω. The witness ω is *infeasible* if there is a path from r' to x in G. Intuitively, for ω to be feasible, all the views must be preserved until the events in ω in the interleaving τ. If (w', r') is broken in τ then r' is not view preserving. Otherwise ω is deemed infeasible in G'. The following theorem provides the necessary condition for feasibility.

Theorem 1: [WITNESS ORDER GRAPH THEOREM] A witness is infeasible if there is a cycle in G'.

A proof sketch is provided in the appendix. The reverse direction (infeasibility⇒cycle) is not true. This has to do with the ordering choice between unordered locked scopes and is considered next.

Consider a pair of locked scopes $[a_1, \ldots, b_1]_l$ and $[a_2, \ldots, b_2]_l$ in different threads guarded by the same lock variable l, such that there does not exist a path from a_1 to b_2 or from a_2 to b_1 in G'. In this case the locked scopes are defined to be an *unordered* pair of locked scopes. Moreover due to the mutual exclusion between the two locked scopes one must be ordered before the other. Thus, there exists a *choice* between edges (b_1, a_2) and (b_2, a_1). The edges (b_1, a_2) and (b_2, a_1) are defined as *choice edges* and the pair $\{(b_1, a_2), (b_2, a_1)\}$ is a *choice edge pair*.

Consider G' shown in Fig. 5. Let there be a witness order edge from y to x (not shown in Fig. 5 for clarity). Let $[a_1, \ldots, b_1]_{l_1}$ and $[a_2, \ldots, b_2]_{l_1}$ be an unordered pair of locked scopes guarded by variable l_1. Similarly, let $[a_3, \ldots, b_3]_{l_2}$ and $[a_4, \ldots, b_4]_{l_2}$ be an unordered pair of locked scopes guarded by variable l_2. Let e_1 and e_2 be choice edges $e_1 \in\{(b_1, a_2),(b_2, a_1)\}$ and $e_2 \in\{(b_3, a_4),(b_4, a_3)\}$. Let the edges shown in Fig. 5 represent paths in G'. *For finding a feasible witness, we need* *at least one combination of choice edges e_1 and e_2 such that their addition to G' leads to no cycle.* In this example, every combination of e_1 and e_2 results in a path from x to y. This combined with the witness order edge (y, x) leads to a cycle for each combination. In general, if there are d choice edge pairs then we need to check 2^d combinations in conjunction with G'. The number of combinations that actually need to be considered can be reduced as shown in the next subsection.

We would like to point out that this example also illustrates that UCG analysis [23] is incomplete in general, since it does not consider choice edges that may result in cycles with more than two threads.

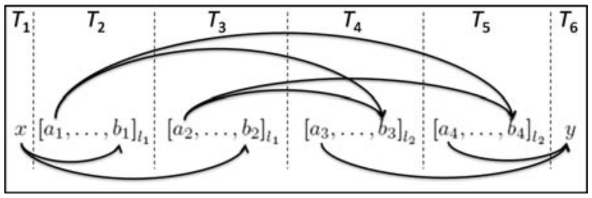


Figure 5. All combinations of choice edges e_1 and e_2 give a path from x to y, where $e_1 \in \{(b_1, a_2),(b_2, a_1)\}$ and $e_2 \in \{(b_3, a_4),(b_4, a_3)\}$

B. Sufficient Condition for Witness: Choice Graph

We first define a *lock abstraction graph* denoted as $G''_a(\omega)$. (Henceforth, we refer to the lock abstraction graph as G''_a when ω is clear from the context.) All vertices within a locked scope in G' are replaced by a single meta-vertex in G''_a. Any edge originating from or terminating into the locked scope, originates from or terminates into the meta-vertex, respectively. Further, for each unordered pair of locked scopes present in G', an undirected edge connects the corresponding meta-vertices in G''_a and is referred to as the *abstract choice edge*. The abstract choice graph for the example shown in Fig. 5 is shown in Fig. 6(a). The vertices $m_1, \ldots, m_4$ are the meta-vertices and the undirected edges (m_1, m_2) and (m_3, m_4) represent the abstract choice edges in G''_a.

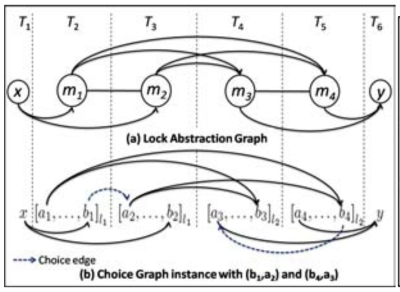


Figure 6. (a) Lock abstraction graph for the example shown in Fig 5. (b) One of the choice graphs with choice edges (b_1, a_2) and (b_4, a_3).

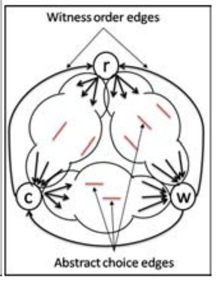


Figure 7. The undirected edges shown in $G''_a(\omega)$ are the abstract choice edges that constitute S_{choice} for witness $w \mapsto c \mapsto r$.

We compute S_{choice} as the set of choice edge pairs such that their exploration is sufficient to detect feasibility of ω. We construct S_{choice} by collecting all the abstract choice edges present in all paths from x to y in G''_a, for all x and y, where (y, x) is a witness order edge in G'. Fig. 7 illustrates this for a witness $w \mapsto c \mapsto r$. Let $|S_{choice}| = d'$. Usually $(d' << d)$. This reduction can be viewed as a form of witness-based slicing of G''_a.

Next, we define the choice graph $G''(\omega)$ as follows. (Henceforth, we refer to the choice graph as G'' when ω is clear from the context.) The vertex set $V(G'') = V(G')$. The edge set $E(G'')$ is $E(G')$ augmented with exactly one choice edge per choice edge pair in S_{choice}. Formally,

$$E(G'') = E(G') \cup \left(\bigcup_{\substack{\forall \{e_{c1}, e_{c2}\} \in S_{choice},\\ e_c \in \{e_{c1}, e_{c2}\}}} \{e_c\} \right)$$

For instance, in the example shown in Fig 5, there are two choice edge pairs that belong to S_{choice}: $\{(b_1, a_2),(b_2, a_1)\}$ and $\{(b_3, a_4),(b_4, a_3)\}$. Each choice graph must choose exactly one edge from each pair. As there are 2^2 combinations possible, there exist four choice graphs for this example. Fig. 6(b) shows one of those choice graphs with a choice edge combination (b_1, a_2) and (b_4, a_3).

Theorem 2: [CHOICE GRAPH THEOREM] A witness is infeasible iff all the choice graphs have cycles.

A proof-sketch is provided in the appendix.

C. The Nondeterminism Checking Algorithm

We now summarize the overall algorithm. We first compute the set of possible witnesses, based on challengers for each read event in a trace. For each such witness ω, in the first phase of our analysis, we construct the witness order graph $(G'(\omega))$ and check for a cycle. The witness is infeasible if there is a cycle in $G'(\omega)$. However, if there is no cycle we proceed to the second phase of our analysis. We compute the set S_{choice}. If S_{choice} is empty, the witness is feasible. Otherwise, we construct $2^{d'}$ choice graphs, where $|S_{choice}| = d'$, and check for a cycle until we find a choice graph with no cycle. If an acyclic choice graph exists, the witness is declared feasible. If all choice graphs contain cycles, then the witness is declared infeasible. *In practice, we need to explore only a handful (mostly one) of these choice graphs to find one without a cycle.*

The complete algorithm is shown in Fig. 8. It generates all feasible witnesses of nondeterminism for a given interleaving ρ. Let, x_i, $i = 1 \ldots m$ be the shared variables in the observed trace ρ. Further, for each shared variable x_i, let L_{x_i} be the list of read-couples, i.e. $L_{x_i} = \{(w, R(w)) \mid w \text{ writes } x_i\}$.

Optimization: Note that the partial order edges $(V(G))$ and all the induced edges due to locks and read-couple edges except (w, r) are present in all the choice graphs for a given witness ω. Therefore, we add all the read-couple edges and

```
**ReportFeasibleWitnesses** (interleaving $\rho$)
1. Construct partial order graph $G$ from $\rho$
2: Visit each vertex and if it accesses shared variable $x_i$
    a. label vertex with locked scopes.
    b. populate $L_{x_i}$.
3: for each $L_{x_i}$, ($i = 1 \ldots m$)
4:  for each write $w_j$ in $L_{x_i}$
5:   for each read $r_k \in R(w_j)$
6:    for each write $c_l$ in $L_{x_i}$ such that $w_j \neq c_l$
7:     Let $(w_j, r_k, c_l)$ be the triplet
8:     for each possible witness $\omega$ for $(w_j, r_k, c_l)$
       //Witness Order Graph Check
9:      Construct $G'(\omega)$ and check for cycle in $G'(\omega)$.
10:     If cycle found in $G'(\omega)$, report $\omega$ is infeasible.
11:     Else construct $G''_a(\omega)$ and compute $S_{choice}$.
12:     Report feasible witness if $S_{choice}$ is empty.
13:     Construct choice graphs until acyclic graph is found
        and report $\omega$ is feasible.
14:     If all choice graphs are cyclic, report $\omega$ is infeasible.
```

Figure 8. Algorithm for reporting feasible witnesses

their induced edges to G before line 3 in Fig. 8. Next, for each witness we do the following: (1) delete the read-couple (w, r) and the appropriate induced edges corresponding to the read-couple (w, r), and, (2) insert the witness order edges and the edges induced by them to produce $G'(\omega)$. Moreover, we use vector clocks [24] for keeping track of the causality relationships necessary for incremental addition or removal of an induced edge.

Complexity Analysis: The symbols introduced for complexity analysis are described in Fig. 9. The complexity of step 1 in procedure **ReportFeasibleWitnesses** is $O(M + N) = O(M)$ as $N \leq M$. The locked scope analysis requires two passes over the trace to label each read/write event with eid's of acquire/release lock events guarding the event. This is $O(m(p + q)L)$. Populating L_{x_i}, for $i = 1 \ldots m$ requires one pass over the trace ($O(N)$). Next we consider the complexity for a single witness. To construct G', we add the read-couple edges ($O(mp)$) and the witness edges ($O(1)$). The induced edges order the locked scopes. Therefore, the number of induced edges added is $O(L^2)$. The number of read-couples in G is $O(mp)$. Thus, $|E(G')| = O(M + mp + L^2)$. Then cycle checking in G' is $O(M + N + mp + L^2) = O(M + L^2)$ (since $N \leq M$ and $mp \leq N$). The number of witnesses is $O(mpq^2)$. Note that in our implementation, the construction of G' is done between step 2 and step 3 of procedure **ReportFeasibleWitnesses** for efficiency, with some simple book-keeping which is omitted here for brevity. Since, the number of choice edge pairs (d) is $O(L^2)$, $d' = O(L^2)$. Therefore, the number of choice graphs is $O(2^{L^2})$. Checking a cycle in a choice graph is $O(M + L^2)$. Therefore, the overall complexity: $O(M + m(p+q)L + mp + mpq^2(M + L^2) + 2^{L^2}(M + L^2)) = O(mpL + mqL + (mpq^2 + 2^{L^2})(M + L^2))$.

Symbol description	Symbol
Number of vertices in G	N
Number of edges in G	M
Number of lock events in G	L
Number of variables	m
Max. number of reads per variable	p
Max. number of writes per variable	q

Figure 9. Symbol table

IV. Results

We have implemented our technique in a prototype tool. This tool is capable of logging/analyzing execution traces generated by both Java programs and multi-threaded C/C++ programs using pthreads. The program traces used are all available online [18]. The C++ benchmark is available online [16]. All the Java benchmarks are publicly available [12], [14], [17], [19], [31]. These traces are manually chosen aiming to have a good mix with respect to graph size and degree of communication between threads.

The tool logs execution traces at runtime from C++ source code instrumented using the commercial front end from Edison Design Group (EDG). For Java programs, we used execution traces logged at runtime by a modified Java Virtual Machine (JVM). For each test case, we first executed the program using the default OS thread scheduling and logged the execution trace. Next we applied our algorithm to detect the feasible witnesses. The graphs are stored in explicit-state form to facilitate cycle checking. The number of vertices in partial order graphs ranged between 100-26000 and the number of edges in those graphs ranged between 150-31000. We would like to highlight here that we originally implemented exploration of the combination of choice edges using an SMT solver, but the cost was prohibitive, failing to finish on several benchmarks. This motivated our current purely graph-based approach.

All our experiments were conducted on an Intel i7 machine (2.67 GHz, 3 GB memory) running Ubuntu 2.6.31-14-generic. Detailed experimental results are reported in the appendix (Table A1) and a summary is presented in Table I.

We make the following observations.

- In 9 out of 25 traces, Phase I alone was sufficient (row 1 in Table I) for our analysis.
- Around 80% of the witnesses in the majority of the traces are found to be infeasible due to the presence of a cycle in the witness order graph G' (Column 4). Since this is a quick check, most of the witnesses are handled quite expeditiously.
- Among the remaining witnesses, a majority of them do not have choice edges ($\sim$17% of the total witnesses) (Column 5). For the traces in row 2, $\sim$3% of the witnesses have choice edges to be explored (Column 6).
- For the witnesses left with choice edges, even when the average number of possible choice graphs per witness is large (Column 7), the number of choice graphs actually explored per witness is close to 1 (Column 8).[4] This is because the exploration stops as soon as an acyclic choice graph is detected. Thus, overall the average number of graphs explored per witness is very close to 1 also (Column 12).

[4]However, 89 of those witnesses were found to be infeasible, i.e., all choice graphs for these witnesses are cyclic.

Table I
SUMMARY OF THE EXPERIMENTAL DATA ON THE WITNESSES OF NONDETERMINISM IN TRACES OF MULTI-THREADED PROGRAMS.

1.	2.	3.	Witness Order Graph Analysis			Choice Graph Analysis		9.	10.	11.	12.
Categories	#Benchmarks	#Possible witnesses	4. Witnesses with cycles in G' (infeasible) (%)	5. Witnesses with no choice edges (feasible) (%)	6. Witnesses with choice edges (%)	7. Possible choice graphs per witness in column 6	8. Choice graphs explored per witness in column 6	Total time taken (sec)	Total feasible witnesses (%)	Total infeasible witnesses (%)	Avg. number of graphs analyzed per witness in column 3
Phase I sufficient	9	104178	85789 (82.35)	18389 (17.65)	0 (0)	–	–	44.3	18389 (17.65)	85789 (82.35)	1
Both phases required	16	5604552	4477107 (79.88)	943516 (16.84)	183929 (3.28)	7.53	1.03	8597	1127356 (20.11)	4477196 (79.88)	1.001

- The time required for witness order graph analysis is much lower than that of choice graph analysis.

V. Related Work

We have already discussed the broad categories of efforts in detecting dataraces and nondeterminism in Section I (Figure 2). We highlight specific related aspects below.

Datarace detection: Broadly, the approaches can be classified into three groups – (1) monitoring [33], [11], [13], [9], (2) predictive analysis [7], [8], [35] and (3) static analysis [36], [20], [22]. Like many of these techniques, we too use happens-before analysis and reasoning about locks. However, our focus is on detecting nondeterminism that is related to, but distinct from, datarace detection. Specifically, we do not have to provide witnesses with unsynchronized memory accesses, which may involve subtle reasoning about locks, e.g. by using lock acquisition histories [21] or causally-precedes relationships [35]. Rather, we consider witnesses with all possible orderings of related events (w, r, and c), where lock reasoning is used only to ensure mutual exclusion. We use a simple notion of lock scopes to enforce mutual exclusion. Chen et al. [8] used a related notion called lock atomicity sets, but they provide a richer abstraction (lock atomicity equivalence) for their purpose of predicting sound interleavings. UCG-based analysis [23] also used cycle-based infeasibility checks, but their analysis is incomplete for more than two threads where choice edges need to be considered. Our lock abstraction graph can be used to identify choice edge pairs in witness-based slicing for other checkers that may use UCG analysis.

Nondeterminism detection: Ensuring deterministic programs has received a lot of attention lately [5]. Vechev et al. proposed a static analysis for verifying determinism in structured parallel programs, based on checking non-overlapping memory accesses in parallel sections [38]. There is some work on specification and dynamic checking for determinism also [6], [32]. Burnim et al. proposed an assertion framework for specifying that programs should behave deterministically and used it to detect nondeterministic behavior [6]. Sadowski et al. proposed a new non-interference specification for deterministically-parallel code, and used a dynamic analysis tool called SideTrack to enforce it [32]. Many other efforts focus on adding synchronization or deterministic scheduling to preempt nondeterministic behavior or related bugs. Vaziri et al. associate synchronization constraints with fields of a class in object-oriented programs, and use static analysis to automatically infer synchronization points to avoid concurrency-related bugs [37]. Navabi et al. insert lightweight synchronization primitives at potential violation points [27]. DThreads replaces the `pthreads` library with an efficient deterministic multi-threading system [25]. CoreDet is a compiler and runtime system for general-purpose software deterministic multi-threading [3]. Other such systems are Determinator [1], Kendo [29] and dOS [4]. In contrast to these efforts, our work does not target specifying or enforcing determinism, but only to check it under standard synchronization and scheduling semantics. Any enforcements (using synchronization or deterministic thread scheduling) can be easily accounted for by adapting the partial orders we consider in our analysis. To the best of our knowledge, our work is the first to use predictive analysis for detecting nondeterminism.

VI. Conclusion

We have proposed a graph-based predictive analysis method for detecting nondeterminism in multi-threaded programs. We analyze each read-couple with all other writes to the same shared variable and determine the conditions for nondeterminism. When these conditions are satisfied, we generate a witness of nondeterminism. Further, we ensure no false positives by ensuring that our witness is feasible, i.e. there exists an interleaving where this witness will be observed. A key property of our method is that we provide a sound and complete[5] predictive technique that explores a reduced set of sufficient interleavings, thereby ensuring that it is efficient. Our experimental results demonstrate the effectiveness of our proposed method on several C/C++ and Java benchmark programs.

Acknowledgment

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program.

[5]complete for predicting from the given trace, not for the entire program

References

[1] Aviram, A., chun Weng, S., Hu, S., Ford, B.: Efficient System Enforced Deterministic Parallelism. In: OSDI (2010)

[2] Ball, T., Burckhardt, S., de Halleux, J., Musuvathi, M., Qadeer, S.: Deconstructing Concurrency Heisenbugs. In: ICSE. pp. 403–404. IEEE (2009)

[3] Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: CoreDet: A Compiler and Runtime System for Deterministic Multi-threaded Execution. In: ASPLOS. pp. 53–64 (2010)

[4] Bergan, T., Hunt, N., Ceze, L., Gribble, S.D.: Deterministic Process Groups in dOS. In: OSDI. pp. 1–16. OSDI'10 (2010)

[5] Bocchino, Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel Programming Must Be Deterministic By Default. In: HotPar. pp. 4–4. HotPar'09 (2009)

[6] Burnim, J., Sen, K.: Asserting and Checking Determinism For Multi-threaded Programs. Commun. ACM 53 (Jun 2010)

[7] Chen, F., Rosu, G.: Parametric and Sliced Causality. In: CAV. pp. 240–253 (2007)

[8] Chen, F., Serbănută, T., Rosu, G.: jPredictor: A Predictive Runtime Analysis Tool for Java. In: ICSE. pp. 221–230 (2008)

[9] Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In: PLDI. pp. 258–269. PLDI '02 (2002)

[10] Cui, H., Wu, J., Gallagher, J., Guo, H., Yang, J.: Efficient Deterministic Multithreading Through Schedule Relaxation. In: SOSP. pp. 337–351. SOSP '11 (2011)

[11] Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A Race-Aware Java Runtime. Commun. ACM 53 (Nov 2010)

[12] Farchi, E., Nir, Y., Ur, S.: Concurrent Bug Patterns and How to Test Them. In: IPDPS. p. 286 (2003)

[13] Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: PLDI. PLDI '09 (2009)

[14] Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In: SPIN. pp. 245–264 (2000)

[15] Havender, J.W.: Avoiding deadlock in multitasking systems

[16] http://incubator.apache.org/thrift/:

[17] http://research.microsoft.com/qadeer/cav_issta.htm: Joint CAV/ISSTA special event on specification, verification, and testing of concurrent software

[18] http://www.princeton.edu/∼sinha/FMCAD12_Traces.zip:

[19] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html: Java grande forum benchmark suite

[20] Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: CAV. pp. 226–239. Springer (2007), LNCS 4590

[21] Kahlon, V., Ivancic, F., Gupta, A.: Reasoning About Threads Communicating via Locks. In: Computer Aided Verification. pp. 505–518 (2005), LNCS 3576

[22] Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic Reduction of Thread Interleavings in Concurrent Programs. In: TACAS. TACAS '09 (2009)

[23] Kahlon, V., Wang, C.: Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In: CAV. pp. 434–449. Springer (2010)

[24] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7) (1978)

[25] Liu, T., Curtsinger, C., Berger, E.D.: DThreads: Efficient Deterministic Multithreading. In: SOSP. pp. 327–336. SOSP '11 (2011)

[26] Mcdowell, C.E., Helmbold, D.P.: Debugging Concurrent Programs. ACM Computing Surveys 21, 593–622 (1989)

[27] Navabi, A., Zhang, X., Jagannathan, S.: Quasi-Static Scheduling For Safe Futures. In: Chatterjee, S., Scott, M.L. (eds.) PPoPP. pp. 23–32. ACM (2008)

[28] Netzer, R.H.B., Miller, B.P.: What Are Race Conditions?: Some Issues and Formalizations. ACM Lett. Program. Lang. Syst. 1 (March 1992)

[29] Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: Efficient Deterministic Multithreading in Software. SIGPLAN Not. 44, 97–108 (Mar 2009)

[30] Papadimitriou, C.H.: The Serializability of Concurrent Database Updates. J. ACM 26(4), 631–653 (1979)

[31] von Praun, C., Gross, T.R.: Static Detection of Atomicity Violations in Object-Oriented Programs. Object Technology 3(6) (2004)

[32] Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In: ESOP. ESOP '09 (2009)

[33] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)

[34] Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive Analysis for Detecting Serializability Errors through Trace Segmentation. In: MEMOCODE (2011)

[35] Smaragdakis, Y., Evans, J., Sadowski, C., Flanagan, J.Y.C.: Sound Predictive Race Detection in Polynomial Time. In: POPL (2012)

[36] Sterling, N.: WARLOCK - A Static Data Race Analysis Tool. In: USENIX Winter. pp. 97–106 (1993)

[37] Vaziri, M., Tip, F., Dolby, J.: Associating Synchronization Constraints With Data In An Object-Oriented Language. In: POPL (2006)

[38] Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic Verification of Determinism For Structured Parallel Programs. In: SAS. SAS'10 (2010)

Appendix

A. Detailed Experimental Results

The detailed experimental results for a sample of traces are given in Table A1. For each benchmark, column 1 presents various statistics of the logged program traces: threads (thrds), number of events (evs), number of lock events (l-evs) and lock variables (l-vars), number of read/write events (rw-evs) and shared variables (rw-vars) and number of wait-notify events (wn-evs). Column 2 shows the total number of possible witnesses in the observed trace. Columns 3-5 and 6-7 show the results of analyses based on witness order graphs and choice graphs, respectively. For the witness order graphs, we report the number of infeasible witnesses (i.e. cycle found) (column 3), number of feasible witnesses (no choice edges and no cycle) (column 4) and witnesses left with choice edges (column 5). Similarly, for the choice graphs, we report the number of possible choice graphs per witness in column 5 that have choice edges (column 6) and number of choice graphs explored per witness in column 5 that have choice edges (column 7). Column 8 shows the total time taken for the analysis. Columns 9 and 10 show the total feasible witnesses and the total infeasible witnesses, respectively. Column 11 reports the average number of graphs analyzed per witness in column 2.

B. Proof Sketch of Theorem 1

Proof Sketch of Witness Order Graph Theorem:

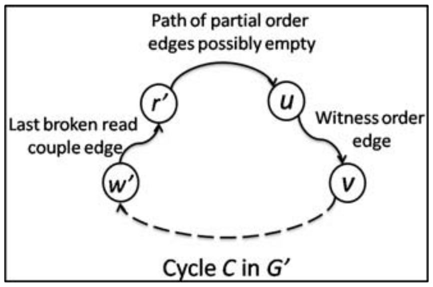

Path of partial order edges possibly empty
Last broken read couple edge
Witness order edge
r′
u
w′
v
Cycle C in G′

Figure B1. Case 2 of the proof of Theorem 1

Let C be the cycle in G'. The partial order edges/induced edges led by partial order edges and read-couples/induced edges led by read-couples only cannot constitute C[6] (otherwise this contradicts the total order of ρ). Therefore, C must contain a witness order edge.

Case 1: C contains witness order edge and partial order edge/induced edge led by partial order edge only. C does not contain read-couple/induced edge led by read-couple: Let τ be an interleaving that contains ω. Due to C, τ is cyclic. Then there does not exist a valid interleaving τ (since it must be a total order of events) that contains ω. Hence, ω is infeasible.

Case 2: C contains witness order edge, partial order edge/induced edge led by partial order edge and at least one read-couple/induced edge led by read-couple: In interleaving τ, at least one read-couple in C must be broken for τ to be a total order since the witness order edges must be observed for τ to be a witness interleaving. What we now need to show is that such broken read couple can alter program flow so that some event x, where x is an event of ω, may not occur. Thus ω will be infeasible.

Let (w', r') be the last read-couple that is broken in C before a witness order edge or an edge induced by a witness order edge and let (u, v) be such an edge in C after (w', r') (Fig. B1). Note that there cannot be any unbroken read-couple (w'', r'') between (w', r') and (u, v) in C because all such reads after a broken read are not guaranteed to happen.

From the construction of (u, v) we know there are two possible cases.

1) The edge (u, v) is an witness order edge: In this case, u is an event in the witness ω. Since the read-couple (w', r') is broken and there are only partial order edges between r' and u, u is not guaranteed to happen in τ, and thus ω is infeasible.
2) The edge (u, v) is induced by an witness order edge: In this case, there is a vertex x, where $x \in \omega$ and $[x \ldots u]_l$, i.e. x and u are in the same locked scope. Since there is a path through partial order edges from r' to u and (w', r') is broken, u may not occur. If u does not occur, then a) either the entire scope $[x \ldots u]_l$ is not executed, in which case ω is infeasible as x is an event in ω, or b) x occurs, but u does not occur and thus the witness ω cannot continue along (u, v). Thus ω is infeasible.□

[6]either a partial order edge or a read-couple edge leads to an induced edge

C. Proof Sketch of Theorem 2

Proof Sketch of Choice Graph Theorem: ($\Leftarrow$) All choice graphs represent traces that are consistent with the witness order graph $G'(\omega)$. We know that G' does not have a cycle, otherwise it would have been detected before. In a choice graph, all the unordered pairs of locked scopes represented in S_{choice} are ordered. The presence of cycle in a choice graph $G''(\omega)$ implies that the witness ω is infeasible with respect to the particular ordering of locked scopes present in $G''(\omega)$. Similarly, the presence of cycles in all choice graphs implies that the witness is infeasible with respect to all the orderings of locked scopes represented in S_{choice}. Hence, ω is infeasible.

($\Rightarrow$) It is known that G' is acyclic (otherwise it would have been detected earlier). Therefore, ω is infeasible implies that there does not exist an (acyclic) interleaving τ that is consistent with any of the 2^d combinations of choice edges. Then all those 2^d graphs are cyclic. The cycles in these 2^d graphs can be divided into two categories, (1) cycles that do not contain any choice edge outside S_{choice}, and, (2) cycles that contain at least one choice edge outside S_{choice}. All cycles of the first category are present in $2^{d'}$ choice graphs. We are done if we can prove that (1) there is no cycle in the second category, and, (2) each of the choice graphs contain at least one cycle from the first category.

Subproof 1: We prove by contradiction. Let the witness be infeasible and there exists a cycle C in one of 2^d possible graphs that contains at least one choice edge e_{c1} from a choice edge pair $t = \{e_{c1}, e_{c2}\}$ outside S_{choice}. C must contain at least one witness order edge (y, x) (otherwise ρ is inconsistent). This choice edge e_{c1} in C is on the path between x and y. Therefore, by definition the choice edge pair t must be within S_{choice} leading to contradiction.

Subproof 2: We prove by contradiction. Let the witness ω be infeasible and there exists an acyclic choice graph G''. This implies that the particular combination of d' choice edges present in G'' does not lead to a cycle (since, by subproof 1, we know that there does not exist a cycle in 2^d combinations that contain choice edges from pairs outside S_{choice}). Then there exists an (acyclic) interleaving τ consistent with choice graph G'' containing ω. Then ω is feasible. This leads to the contradiction.

Hence, if the witness is infeasible, then all the choice graphs must have cycles in them.□

Table A1
EXPERIMENTAL DATA ON THE WITNESSES OF NONDETERMINISM IN TRACES OF MULTI-THREADED PROGRAMS.

1. Benchmark	2. #Possible witnesses	Witness Order Graph Analysis: 3. Witnesses with cycles in G' (infeasible) (%)	4. Witnesses with no choice edges (feasible) (%)	5. Witnesses with choice edges (%) (ϵ=∼0)	Choice Graph Analysis: 6. Possible choice graphs per witness in column 5	7. Choice graphs explored per witness in column 5	8. Total time taken	9. Total feasible witnesses (%)	10. Total infeasible witnesses (%)	11. Avg. number of graphs analyzed per witness in column 2
conpool - thrds: 4, evs: 97, l-evs: 16, l-vars: 1, rw-evs: 53, rw-vars: 5, wn-evs: 3	252	221 (88)	31 (12)	0 (0)	–	–	0.007s	31 (12.3)	221 (87.7)	1
liveness - thrds: 7, evs: 283, l-evs: 44, l-vars: 9, rw-evs: 163, rw-vars: 12, wn-evs: 6	855	709 (83)	146 (17)	0 (0)	–	–	0.064s	146 (17)	709 (83)	1
SynchBench - thrds: 16, evs: 1510, l-evs: 306, l-vars: 2, rw-evs: 533, rw-vars: 15, wn-evs: 0	47526	39474 (83)	8052 (17)	0 (0)	–	–	8.85s	8052 (17)	39474 (83)	1
Barrier - thrds: 10, evs: 653, l-evs: 108, l-vars: 2, rw-evs: 262, rw-vars: 12, wn-evs: 7	3975	3231 (81)	744 (19)	0 (0)	–	–	0.62s	744 (18.7)	3231 (81.3)	1
account - thrds: 11, evs: 902, l-evs: 146, l-vars: 21, rw-evs: 430, rw-vars: 42, wn-evs: 10	1416	1042 (73.6)	326 (23)	48 (3.4)	36.33	2.83	1.352s	374 (26.1)	1058 (73.9)	1.06
DaisyTest - thrds: 3, evs: 2998, l-evs: 422, l-vars: 10, rw-evs: 2003, rw-vars: 45, wn-evs: 15	383007	305635 (79.8)	61852 (16.1)	15520 (4.1)	6.35	1.12	244s	77372 (20.2)	305650 (79.8)	1.005
Elevator - thrds: 4, evs: 3004, l-evs: 370, l-vars: 11, rw-evs: 1795, rw-vars: 70, wn-evs: 0	3249	2671 (82)	578 (18)	0 (0)	–	–	0.8s	578 (17.8)	2671 (82.2)	1
philo - thrds: 6, evs: 1141, l-evs: 126, l-vars: 6, rw-evs: 857, rw-vars: 23, wn-evs: 22	4893	4118 (84)	775 (16)	0 (0)	–	–	0.65s	775 (15.8)	4118 (84.2)	1
ThriftTrace1 - thrds: 4, evs: 2406, l-evs: 226, l-vars: 12, rw-evs: 869, rw-vars: 62, wn-evs: 53	618	361 (58)	96 (16)	161 (26)	105.6	1	1.6s	257 (41.6)	361 (58.4)	1
ThriftTrace2 - thrds: 4, evs: 11357, l-evs: 1384, l-vars: 48, rw-evs: 3184, rw-vars: 171, wn-evs: 324	35610	29441 (82.7)	5804 (16.3)	365 (1)	296.7	1	28s	6169 (17.3)	29441 (82.7)	1
ThriftTrace3 - thrds: 6, evs: 20640, l-evs: 1724, l-vars: 158, rw-evs: 8818, rw-vars: 519, wn-evs: 349	479142	399814 (83)	79326 (16.5)	2 (ϵ)	18	1	243s	79328 (16.6)	399814 (83.4)	1

Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)

Automatic Lock Insertion in Concurrent Programs

Vineet Kahlon, NEC Labs, Princeton, USA.

***Abstract*—Triggering errors in concurrent programs is a notoriously dif cult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of operations of different threads. An even more challenging task is xing errors once they are detected. In general, automatically synthesizing a correct program from a buggy one is a hard problem. However for simple correctness properties that depend on the syntactic structure of the program rather than its semantics, automatic error correction becomes feasible. In this paper, we consider the problem of lock insertion to enforce critical sections required to x bugs like atomicity violations. A key challenge in lock insertion is that enforcing critical sections is not the sole criterion that needs to be satis ed. Often other correctness constraints like deadlock-freedom also need to be met. Moreover, apart from ensuring correctness, another key concern during lock insertion is performance. Indeed, mutual exclusion constraints generated by locks kill parallelism thereby impacting performance. Thus it is crucial that the newly introduced critical sections be kept as small as possible. In other words, our goal is lock insertion while meeting the dual, and often con icting, requirements of (i) correctness and (ii) performance. In this paper, we present a fully automatic, provable optimal, ef cient and precise technique for lock insertion in concurrent code that ensures deadlock freedom while attempting to minimize the resulting critical sections.**

I. Introduction

Detecting errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of different threads. An even more challenging task is fixing errors once they are detected. In general, automatically synthesizing a correct program from a buggy one is hard. However for simple correctness properties that depend on the syntactic structure of the program rather than its semantics, automatic error correction becomes feasible. An example is the insertion of mutexes in order to enforce critical sections to fix data races or atomicity violations. Inserting mutexes typically does not require reasoning about program semantics but relies merely on aliasing information in order to identify sections of code with shared variable accesses that need to be executed atomically.

In this paper, we consider the problem of lock insertion to enforce critical sections required to fix bugs like atomicity violations. This can be accomplished in a trivial manner by simply encapsulating the desired regions of code within lock/unlock statements. However, enforcing critical sections is often not the sole criterion to be satisfied during lock insertion. Indeed, adding mutexes may introduce new deadlocks. Thus a key goal is to guarantee deadlock-free lock insertion, i.e., no *new* deadlocks are introduced.

Apart from ensuring correctness, another key concern during lock insertion is performance. Mutual exclusion constraints generated by locks kill parallelism thereby impacting performance. Thus it is critical that the newly introduced critical sections be kept as small as possible.

It is worth mentioning that there exist techniques in the literature [4], [12], [6], [1], [2] for lock insertion in programs without prior locks. However, this problem is easier than the one we consider in this paper, as for programs without locks deadlocks can be avoided simply by acquiring all locks in a pre-defined order. One way to handle lock insertion in programs with prior locks would be to first remove all pre-existing locks and then leverage existing lock insertion techniques. This approach, however, presents many practical obstacles.

First, before removing existing locks we would have to identify all pairs of mutually atomic segments of the form (s_1, s_2), where atomic segments s_1 and s_2 are guarded by the same lock. However, lock/unlock APIs typically take pointers to locks as parameters and so a whole program points-to analysis would be required in order to determine the locks guarding segments s_1 and s_2. Moreover, since lock pointers often point to different locks in different function calling contexts, this points-to analysis needs to be context-sensitive. However, it is well known that scaling a precise context-sensitive points-to analysis for large realistic programs comprised of multiple code modules is a non-trivial task.

Moreover, even after aliases have been computed precisely, it is not enough to enumerate all pairs of the form (s_1, s_2), where s_1 and s_2 are guarded with the same lock. This is because it is often the case that locks are re-used (to reduce their number in certain applications) so that segments s_1 and s_2 may be guarded with the same lock even though they may not execute in parallel. Thus in order to isolate all pairs of segments that are truly mutually atomic, we would need to (1) understand the reasons for introducing prior locking statements, i.e., be somewhat knowledgeable about the program's semantics which is not feasible for large applications, and (2) need at least a whole program MHP (may-happens-in-parallel) analysis to determine whether s_1 and s_2 can execute in parallel - expensive for large programs. Finally, we may end up having a large number of mutually atomic pairs of segments impacting scalability of lock insertion.

On the other hand, it is highly desirable that our lock insertion technique avoids a whole program analysis and restricts the analysis (including the context-sensitive points-to analysis) to only the few modules requiring code modification, i.e., where bugs have been detected. This is precisely what our lock insertion technique accomplishes. An important side benefit is that it ensures scalability of our analysis.

To sum up, our goal is a *localized* analysis for lock insertion meeting the dual constraints of (i) correctness and (ii) performance. These constraints are often conflicting in nature. Indeed, during lock insertion one of the key properties that we want to ensure is deadlock-freedom while keeping the critical sections as small as possible. If either one of these two requirements is dropped, then the problem is greatly simplified. For instance, if we give up the requirement of deadlock freedom then given a pair of code segments s_1 and s_2

to be executed in a mutually atomic fashion, it suffices to insert lock (unlock) statements for a new lock l, immediately before (after) the two segments in both threads. Clearly this induces minimal critical sections but does not guarantee that no new deadlocks have been introduced. Similarly, if the requirement of optimality is dropped then it suffices to introduce lock (unlock) statements for a new lock l at the last lock free states before (after) the segments in either thread. Such a solution ensures that no new deadlocks are introduced but may not be optimal.

Given a pair of mutually atomic code segments s_1 and s_2 in two different threads T_1 and T_2, respectively, of an n-thread program, we present a lock insertion strategy that involves a series of local moves that re-locates the newly inserted lock statements in the individual threads T_1 and T_2 in a dovetailed fashion till we achieve deadlock freedom. The interesting, and somewhat surprising, result is that our objective of minimizing the newly introduced critical sections which is inherently global in nature can be achieved via purely local moves of the locking statements in the individual threads. This is crucial as it allows our strategy to be *compositional* in nature, i.e., based only on thread local reasoning, thereby ensuring scalability.

While our lock insertion strategy is applicable to programs with arbitrary locking patterns, for implementation purposes we consider the special case of programs with *nested* locks. The main motivation for this is that almost all lock usage in real life programs is nested. Additionally, nested locks offer a key advantage in that they allow us to leverage the framework of acquisition histories [10] to formulate a provable efficient and compositional (thread local) analysis for lock insertion.

We demonstrate the efficacy of our technique on a broad range of benchmarks.

II. Program Model

We consider concurrent imperative programs comprised of threads that communicate using shared variables and synchronize with each other using standard primitives such as locks and rendezvous. Formally, we define a concurrent program $\mathcal{CP}$ as a tuple $(\mathcal{T}, \mathcal{V}, \mathcal{R}, s_0)$, where $\mathcal{T} = \{T_1, ..., T_n\}$ denotes a finite set of threads, $\mathcal{V} = \{v_1, ..., v_m\}$ a finite set of shared variables and synchronization objects with v_i taking on values from the set V_i, $\mathcal{R}$ the transition relation and s_0 the initial state of $\mathcal{CP}$. Each thread T_i is represented by the control flow graph of the sequential program it executes, and is denoted by the pair (C_i, R_i), where C_i denotes the set of control locations of T_i and R_i its transition relation. A global state s of $\mathcal{CP}$ is a tuple $(s[1], ..., s[n], v[1], ..., v[m]) \in \mathcal{S} = C_1 \times ... \times C_n \times V_1 \times ... \times V_m$, where $s[i]$ represents the current control location of thread T_i and $v[j]$ the current value of variable v_j. The global state transition diagram of $\mathcal{CP}$ is defined to be the standard interleaved parallel composition of the transition diagrams of the individual threads.

III. Lock Insertion Problem

The goal of lock insertion is to remove data races or, more generally, atomicity violations by enforcing critical sections that envelope regions of code to be executed atomically. These critical section may comprise multiple regions of contiguous code that we refer to as *atomic segments*. Due to branching,

```
T1() {
0a:  ...
1a:  while(sh   > 0) {
2a:  sh++;
3a:  ...
4a:  }
}
```

```
T2() {
0b:  ...
1b:  sh  =  sh  +  2;
2b:  ...
}
```

Fig. 1. Split Critical Section

loops and recursion, a code segment of thread T is, in general, defined by a sub-graph of the CFG of T.

As an example, consider the threads T_1 and T_2 shown in Fig. 1 accessing shared variable sh . In thread T_1, due to the presence of a loop the critical section is broken up into two segments, one comprising the statements 1a and 2a and the other comprising the statement 4a . Note that we need to include 4a in the critical section because the condition of the while loop accesses the shared variable sh and we need to re-acquire the lock guarding access to 1a (in case it was released within the loop body) if we re-enter the loop body. The critical section in thread T_2, however, consists of only one atomic segment, i.e., 1b .

We define an *atomic segment* of thread T as the set of control locations occurring in a directed acyclic graph (DAG) whose (i) *roots*, i.e., nodes of in-degree zero, define the control locations of T marking the start of the segment, (ii) *leaves* define control locations marking the ends of the segment, and (iii) the successors of each location c in the segment are the non-backedge (as defined by some dfs ordering) successors of c in the CFG of T. We use $[(r_1, ..., r_p), (l_1, ..., l_q)]$ to denote an atomic segment with roots $r_1, ..., r_p$ and leaves $l_1, ..., l_q$. For example, $[\mathtt{1a}, \mathtt{2a}]$ (or more precisely $[(\mathtt{1a}), (\mathtt{2a})]$) denotes an atomic segment of T_1 in Fig. 1.

Whether a region of code in a thread is an atomic segment depends on the values of program counters of other threads. Indeed regions of code in different threads accessing the same shared variable need to be executed atomically *relative* to each other, while regions of code accessing different shared variables need not. This leads to the notion of mutually atomic segments.

De nition (Mutually Atomic Segments). *We say that code segments s_1 and s_2 of threads T_1 and T_2, respectively, are mutually atomic if there does not exist a reachable global state of the given concurrent program with T_1 and T_2 at control locations c_1 and c_2 occurring along segments s_1 and s_2, respectively.*

The lock insertion problem is then defined as follows.

Lock Insertion Problem. *Let $P = \{(s_1^1, s_1^2), ..., (s_k^1, s_k^2)\}$ be a set of pairs (s_j^1, s_j^2) of atomic segments s_j^1 and s_j^2. Identify locations in threads $T_1, ..., T_n$ comprising the given concurrent program to insert locks that guarantees the following*

1) *for each j, s_j^1 and s_j^2 are mutually atomic,*
2) *no new deadlocks are introduced, and*
3) *minimality of the newly introduced critical sections, as determined by the set of program statements in the critical sections.*

Conditions 1, 2 and 3 are collectively referred to as *Lock Insertion Requirements*. It is worth pointing out that our notion of minimality for critical sections is based on set inclusion as opposed to the number of program statements comprising the critical section. This is the best one can hope for.

Consistency Invariant. In Fig. 1, we observe that the critical section in T_1, was 'split' into the segments $[\mathtt{1a}, \mathtt{2a}]$ and $[\mathtt{4a}, \mathtt{4a}]$ to maintain the consistency invariant that an un-acquired lock cannot be released (in case we re-enter the loop). We therefore assume that the atomic segments in the specification of the given lock insertion problem instance satisfy the following natural condition.

Consistency Invariant. *Let $P = \{(s_1^1, s_1^2), ..., (s_k^1, s_k^2)\}$ be a lock insertion problem instance, where for each j, s_j^1 and s_j^2 are the desired mutually atomic segments. Then if a loop head (tail) occurs in an atomic segment s_j^m comprising a critical section cs of thread T_i then its matching loop tail (head) also occurs in an (possibly the same) atomic segment $s_{j'}^{m'}$ comprising cs.*

IV. Lock Insertion

We start by observing that it suffices to formulate the lock insertion procedure for the case where we are given a single pair (CS_1, CS_2) of mutually atomic segments, where CS_1 and CS_2 are atomic segments in two different threads. The case where we are given multiple mutually atomic segment pairs can be handled by repeatedly applying the lock insertion procedure.

For ease of exposition, we start with the assumption that the threads are specified as straight line code with the general case being considered in sec V. The straight-line case suffices to show case the key ideas behind our lock insertion technique.

Let the given concurrent program be comprised of the threads $T_1, ..., T_k$ and let atomic segments CS_1 and CS_2 belong to threads T_1 and T_2. Suppose that threads T_1 and T_2 are defined via the sequences of control locations $T_1 : c_0, ..., c_n$ and $T_2 : d_0, ..., d_m$, respectively.

For the case where thread T is specified as the straight-line code $T : d_0, ..., d_p$, a segment s defining a critical section of T can be identified uniquely by its start and end locations d_i and d_j, respectively, where $i < j$. We denote such a segment by $s = [d_i, d_j]$, where $[d_i, d_j]$ denotes the set of control locations occurring between (and including) d_i and d_j along T.

Let the segments s_1 and s_2 of threads T_1 and T_2 be denoted by $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$, respectively, where $i < j$ and $i' < j'$. Our goal is to introduce locking and unlocking statements *lock(l)* and *unlock(l)* for a *new* lock l, respectively, such that the lock insertion requirements are met.

Notation. Before proceeding further, we fix some notation. The locking statements *lock(l)* and *unlock(l)* inserted in threads T_1 and T_2 are abbreviated as l_1 and l_2 whereas the unlocking statements are abbreviated as u_1 and u_2, respectively. If l_1 (l_2) and u_1 (u_2) are added immediately before c_p ($d_{p'}$) and immediately after c_q ($d_{q'}$), respectively, then the resulting critical sections, i.e., the set of statements between (and including) l_1 (l_2) and u_1 (u_2) are denoted by $[\![c_p, c_q]\!]_l$ ($[\![d_{p'}, d_{q'}]\!]_l$).

During lock insertion, two sets of decisions need to be made:

- **Lock Statement Insertion**: determining locations of insertion of the lock statements l_1 and l_2, and
- **Unlock Statement Insertion**: determining locations of insertion of the matching unlock statements u_1 and u_2.

```
T1() {
...
c0:  lock(m);
...
c1:  lock(n);
...
c2:  unlock(n);

   //  begin  critical   section
   local1    =  account_value;
   local1    +=  increment;
   account_value      =  local1;
   // end  critical    section

c3:  unlock(m);
...
}
```

```
T2() {
d0:  lock(n);
d1:  ...

   //  begin  critical   section
   local2    =  account_value;
d2:  lock(m);
   //  access   another   account
   local2   +=  other_account_value;
d3:  unlock(m);
   account_value      =  local2;
   //  end  critical    section

d4:  unlock(n);
}
```

Fig. 2. Lock Insertion Example.

A. Insertion of Unlocking Statements

While determining locations where to insert the locking statements is not straightforward, we observe that since unlock statements are non-blocking they cannot participate in a deadlock. It follows that in order to enforce the mutually atomicity of the segments $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$, it suffices to insert the unlocking statements u_1 and u_2 immediately after c_j and $d_{j'}$, respectively. Formally,

Theorem 1 (Unlock Insertion). *Let $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$ be segments of threads T_1 and T_2, respectively, de ning mutually atomic segments to be enforced. Let $[\![c_a, c_b]\!]_l$ and $[\![d_{a'}, d_{b'}]\!]_l$, where $[c_i, c_j] \subseteq [\![c_a, c_b]\!]_l$ and $[d_{i'}, d_{j'}] \subseteq [\![d_{a'}, d_{b'}]\!]_l$, be critical sections enforcing mutually atomicity of $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$ that also satisfy the lock insertion requirements. Then $c_b = c_j$ and $d_{b'} = d_{j'}$.*

B. Insertion of Locking Statements

We now turn to the more interesting problem of inserting the locking statements l_1 and l_2. If guaranteeing deadlock freedom were not a requirement then inserting the statements l_1 (l_2) immediately before locations c_i ($d_{i'}$) in thread T_1 (T_2), suffices. Clearly, the resulting critical sections $[\![c_i, c_j]\!]_l$ and $[\![d_{i'}, d_{j'}]\!]_l$ satisfy lock insertion requirement 1. Moreover, since by requirement 1, $[c_i, c_j]$ ($[d_{i'}, d_{j'}]$) must belong to any critical section enforced by our newly inserted lock/unlock statements in thread T_1 (T_2), we see that $[\![c_i, c_j]\!]_l$ and $[\![d_{i'}, d_{j'}]\!]_l$ would indeed be minimal (based on set inclusion) critical sections potentially satisfying the lock insertion requirements.

However, inserting l_1 and l_2 immediately before c_i and $d_{i'}$, respectively, could introduce new deadlocks. Consider, for example, the concurrent program $\mathcal{C}$ comprised of threads T_1 and T_2 with the desired critical sections shown in fig 2. The running of our lock insertion procedure is demonstrated on the CFGs of T_1 and T_2 in fig 3. Here the original lock/unlock statements have been shown as black circles while the mutually atomic segments (CS_1, CS_2) to be enforced as rectangles. Let $CS_1 = [c_i, c_j]$ and $CS_2 = [d_{i'}, d_{j'}]$. Inserting l_1 and l_2 (shown as white circles) immediately before c_i and $d_{i'}$ results in the threads shown in Fig. 3(a). Note that at location c_3 thread T_1 holds lock m which was acquired at c_0, whereas at location d_2, thread T_2 holds lock l_2 acquired at d_1. Thus at global control location $(\mathsf{c}_3, \mathsf{d}_2)$ of $\mathcal{C}$, T_1 holds lock m and is waiting at acquire l, whereas T_2 holds l and is waiting to acquire m. This cyclic dependency creates a deadlock.

Thread T_1 move. Recall that our goal is to guarantee deadlock freedom while ensuring minimality of the newly introduced critical sections. Towards that end, we started by inserting the locking statements l_1 and l_2 immediately before c_i and $d_{i'}$, respectively, even if the newly synthesized threads have deadlocks. If no new deadlocks are introduced then we are done. If there exist newly introduced deadlocks, they must involve at least one of l_1 or l_2. In our case, $\mathsf{c}_3 : \mathsf{l}_1$ can potentially be involved in a deadlock but $\mathsf{d}_1 : \mathsf{l}_2$ cannot. Thus in order to guarantee deadlock freedom, we need to re-locate the locking statement l_1. We observe that l_1 cannot be moved forward as that would cause it to enter the critical section CS_1 which we are supposed to enforce. Thus l_1 can only be moved backwards along T_1.

In order to ensure that l_1 does not participate in a deadlock we move l_1 backwards along T_1 till we encounter a control location where it can no longer be involved in any deadlock. In order to identify this location, we recall that two conditions need to be satisfied in order for $c_k : \mathsf{l}_1$ to be involved in a deadlock with a statement $d_{k'} : lock(m)$ of thread T_2.

1) **Reachability:** $(c_k, d_{k'})$ are pairwise reachable, and
2) **Cyclic Dependency:** locks m and l are held at c_k and $d_{k'}$, respectively

Thus in order to identify the location where to introduce l_1 in thread T_1, we keep moving it backwards starting from c_i till we encounter a control location c_k where at least one of the above conditions is falsified. By condition 2, we see that if $c_k : \mathsf{l}_1$ deadlocks with location $d_{k'}$ of T_2, lock l must be held at $d_{k'}$. Thus it follows that the *lock(l)* statement in T_1 can deadlock only with a locking statement in the critical section $[\mathsf{l}_2, \mathsf{u}_2]$ in thread T_2. Motivated by the above observation, we define $L_{[\mathsf{l}_2,\mathsf{u}_2]}$ to be the set of locks p such that a statement of the form *lock(p)* occurs along $[\mathsf{l}_2, \mathsf{u}_2]$.

Let c_k, where $k \leq i$ be the last control location occurring before c_i along T_i such that (i) c_k violates condition 1 or 2, and (ii) for each $r \in [k+1..i]$, c_r does not violate any of the conditions 1 or 2. Then we insert l_1 immediately before c_k. Note that, by our construction, c_k is the first location encountered by traversing backwards along T_1 starting at c_i where a *lock(l)* statement can be inserted without it being involved in a deadlock. In our example, in order to remove all potential deadlocks involving l_1 we move it to location c_4 (see Fig. 3(b)).

Deadlock Check. Having removed the deadlocks involving l_1, we check whether l_2 is involved in a deadlock. If not then the procedure terminates.

Thread T_2 move. If, on the other hand, l_2 is involved in a deadlock we remove deadlocks involving l_2 using the same procedure as above - the only difference being that we now consider the deadlocks involving l_2 and the locks acquired along $[\mathsf{l}_1, \mathsf{u}_1]$. We keep moving l_2 backwards along T_2 till we reach a control location of T_2 where l_2 cannot be involved in a deadlock. In our example, we see that even though $\mathsf{d}_1 : \mathsf{l}_2$ couldn't be involved in a deadlock in the original program Fig. 3(a), in the new program Fig. 3(b) gotten via enlargement of the critical section induced by lock l in T_1, $\mathsf{d}_1 : \mathsf{l}_2$ can potentially deadlock with location c_1. In order to remove deadlocks involving l_2 we re-locate it back to location d_4.

Dovetailing. Note, however, that as we move l_2 backwards along T_2, we enlarge the critical section $[\mathsf{l}_2, \mathsf{u}_2]$. A key consequence is that the enlarged critical section may contain new locking statements which may now induce new deadlocks with l_1. In order to remove these deadlocks we again repeat the above procedure by moving l_1 further backwards till it cannot be involved in a deadlock.

The whole process of removing deadlocks involving statements l_1 and l_2 in a dovetailed fashion, wherein the statements l_1 and l_2 are re-located backwards, is continued till all deadlocks involving l_1 and l_2 are removed. This yields us a deadlock free insertion of l_1 and l_2 in T_1 and T_2, respectively (see Fig. 3(c)).

A formal description of the lock insertion procedure is formulated as Alg. 1.

Algorithm 1 Lock Insertion for Straight-line Code

```
1: Input: Threads $T_1, ..., T_n$ specified as straight-line code,
   with $T_1$ and $T_2$ defined by the sequences $c_0, ..., c_n$ and
   $d_0, ..., d_m$, respectively, and mutually atomic segments
   $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$ of $T_1$ and $T_2$, respectively.
2: Insert $\mathsf{u}_1$ and $\mathsf{u}_2$ in threads $T_1$ and $T_2$ immediately after
   $c_j$ and $d_{j'}$, respectively.        (Insertion of Unlock Statements)
3: Insert $\mathsf{l}_1$ and $\mathsf{l}_2$ in threads $T_1$ and $T_2$ immediately before
   $c_i$ and $d_{i'}$, respectively.
4: repeat
5:   if $\mathsf{l}_1$ can be involved in a potential deadlock then
6:     Move $\mathsf{l}_1$ backward along $T_1$ till we reach a control
       location $c'$ of thread $T_1$ such that for each lock
       $m \in L_{[\mathsf{l}_2,\mathsf{u}_2]}$: either (i) $m$ is not held at $c'$, or (ii)
       for each location $d'$ in critical section $[\mathsf{l}_2, \mathsf{u}_2]$ where
       $m$ is acquired, $c'$ and $d'$ are not pairwise reachable.
7:   end if
8:   if $\mathsf{l}_2$ can be involved in a potential deadlock then
9:     Move $\mathsf{l}_2$ backward along $T_2$ till we reach a control
       location $d'$ of thread $T_2$ such that for each lock
       $m \in L_{[\mathsf{l}_1,\mathsf{u}_1]}$: either (i) $m$ is not held at $d'$, or (ii)
       for each location $c'$ in critical section $[\mathsf{l}_1, \mathsf{u}_1]$ where
       $m$ is acquired, $c'$ and $d'$ are not pairwise reachable.
10:  end if
11: until there do not exist any deadlocks involving $\mathsf{l}_1$ or $\mathsf{l}_2$
```

C. Meeting Lock Insertion Requirements

We now show the somewhat surprising result that simply by making local moves of l_1 and l_2 in a dove-tailed manner as encoded in Alg. 1, all three of our (global) lock insertion requirements are met.

Enforcement of Mutual Atomicity. Since Alg. 1 always maintains the invariants that $[c_i, c_j] \subseteq [\mathsf{l}_1, \mathsf{u}_1]$ and $[d_{i'}, d_{j'}] \subseteq [\mathsf{l}_2, \mathsf{u}_2]$ we see that the mutual atomicity of $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$ is enforced.

Deadlock Freedom. The termination condition (step 11) of Alg. 1 ensures that there do not exist deadlocks involving l_1 or l_2 and since the newly introduced deadlock must involve at least one of these lock statements we see that requirement 2 is also met.

Optimality. The most interesting part is to show that requirement 3 is met, i.e., the critical sections identified by Alg. 1

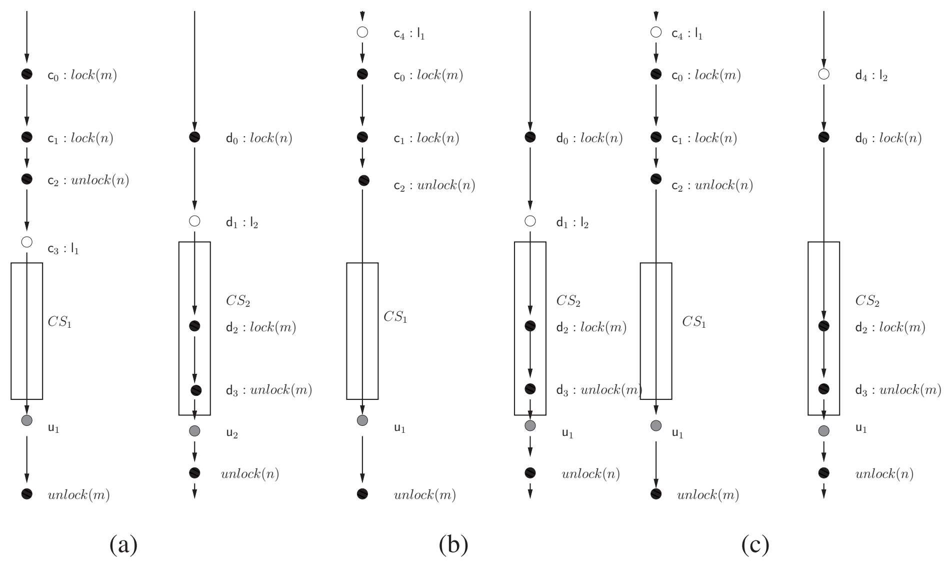


Fig. 3. Lock Insertion Procedure

are optimal. The proof is provided in the full version of the paper [?].

Optimality Result. *Let $[c_a, c_b]$ and $[d_{a'}, d_{b'}]$ be critical sections satisfying the lock insertion requirements. Then $[\mathsf{l}_1, \mathsf{u}_1] \subseteq [c_a, c_b]$ and $[\mathsf{l}_2, \mathsf{u}_2] \subseteq [d_{a'}, d_{b'}]$, where $[\mathsf{l}_1, \mathsf{u}_1]$ and $[\mathsf{l}_2, \mathsf{u}_2]$ are the critical sections in threads T_1 and T_2, respectively, identi ed by Alg. 1.*

Proof.

We prove the result by contradiction. If possible, suppose that $[c_a, c_b]$ is a proper subset of $[\mathsf{l}_1, \mathsf{u}_1]$. As discussed before, the fact that unlock statements are non-blocking combined with the optimality requirement imply that $\mathsf{u}_1 = c_b$ and $\mathsf{u}_2 = d_{b'}$. Then from the assumption that $[c_a, c_b]$ is a proper subset of $[\mathsf{l}_1, \mathsf{u}_1]$ we can deduce that l_1 occurs before c_a along T_1.

For $r \geq 0$, let l_1^r and l_2^r be the locations of the *lock(l)* statements in threads T_1 and T_2 after the rth iteration of Alg. 1. Suppose that k is the largest index for which l_1^k belongs to the interval $[c_a, c_b]$ and l_2^k belongs to the interval $[c_{a'}, c_{b'}]$. At the $(k+1)$st iteration, either l_1 moves out of the interval $[c_a, c_b]$ or l_2 moves out of the interval $[c_{a'}, c_{b'}]$. For definiteness assume that it is the former.

To prove our claim we now show that l_1 cannot move out of the interval $[c_a, c_b]$. Indeed for it to move out, l_1 needs to be propagated backwards along T_1 till it crosses c_a. At the time of crossing c_a, T_1 must be holding a lock m such that (i) the last statement to acquire m occurs before c_a along T_1, and (ii) there exists a lock acquisition statement for m in the critical section $[\mathsf{l}_2^k, \mathsf{u}_2^k]$. Let L_{c_a} be the set of locks held at c_a that are also acquired in the critical section $[\mathsf{l}_2^k, \mathsf{u}_2^k]$. Clearly $L_{c_a} \neq \emptyset$. Furthermore, there exists a lock $m' \in L_{c_a}$ such that $(c_a, d_{m'})$ are pairwise reachable for some statement $d_{m'}$ acquiring lock m' in $[\mathsf{l}_2^k, \mathsf{u}_2^k]$. If that is not the case then l_1^{k+1} would not cross c_a contradicting the maximality of k. However this creates a deadlock involving locations c_a and $d_{m'}$ of T_1 and T_2, respectively. This is because at c_a, thread T_1 holds lock m' and is waiting to acquire lock l (recall that, by definition, c_a is a *lock(l)* statement), whereas at $d_{m'}$ thread T_2 is holding lock l (as $d_{m'}$ lies in the critical section $[\mathsf{l}_2^k, \mathsf{u}_2^k]$) and waiting to acquire m'. Recall that $d_{m'} \in [\mathsf{l}_2^k, \mathsf{u}_2^k] \subseteq [d_{a'}, d_{b'}]$. Thus $[c_a, c_b]$ and $[d_{a'}, d_{b'}]$ do not meet lock insertion requirement 2 contradicting our hypothesis.

Similarly we may show that $[\mathsf{l}_2, \mathsf{u}_2] \subseteq [d_{a'}, d_{b'}]$.

Note that our notion of minimality for critical sections is based on set inclusion as opposed to the number of program statements comprising the critical section. This is the best one can hope for.

V. Lock Insertion: The General Case

Acyclic CFGs We start by considering the case where the CFGs of threads are acyclic. Here each atomic segment is a DAG (directed acyclic graph) with possibly multiple roots (vertices of in-degree zero) and possibly multiple leaves (vertices of out-degree zero). Thus we assume that the input to the procedure is a pair of mutually atomic segments $s_1 = [(c_{i1}, ..., c_{ik}), (c_{j1}, ..., c_{jp})]$ and $s_2 = [(d_{i'1}, ..., d_{i'k'}), (d_{j'1}, ..., d_{j'p'})]$, wherein the first (second) tuple in each segment represents the roots (leaves) of the segment. Generalization of Alg. 1 to DAGs requires little modification as the notion of backwards traversal required for steps 6 and 9 is well defined. The core idea of lock insertion remains the same as for straight-line code. We start by inserting the unlock statement u_1 immediately after the control locations $c_{j1}, ..., c_{jp}$ and the unlock statement u_2 immediately after the control locations $d_{j'1}, ..., d_{j'p'}$ (step 3 of Alg. 2). This step is analogous to the case of straight-line code, the only difference being that instead of a unique *leaf* node there are multiple *leaves*. Again, as for straight-line code, the locking statements l_1 and l_2 are inserted (step 4) immediately before the locations $c_{i1}, ..., c_{ik}$ and the locations $d_{i'1}, ..., d_{i'k'}$, respectively.

As before, in order to remove deadlocks involving l_1 and l_2 we propagate these statements upwards along the CFGs of the respective threads (steps 5-10 of Alg. 2) via Alg. 3.

In propagating the lock statements l_1 and l_2 upwards along the DAGs, there are two main differences from the straight-line case. First due to branching, we may encounter the same control location multiple times. To track the control locations that have already been visited we maintain a set *Visited* and insert a check (step 10 of Alg. 3) to prevent repeat processing. Secondly, we may encounter control locations with multiple predecessors in which case we need to propagate the locking statement backwards along multiple branches (steps 9-17 of Alg. 3).

Cyclic CFGs. In the general case, due to the presence of cycles in the CFG (caused by loops) the notion of backwards traversal is not well-defined. However, we can reduce the problem of lock insertion for cyclic CFGs to the acyclic case. Towards that end, we leverage the consistency assumption (Sec. III) wherein if a loop head (tail) occurs in an atomic segment comprising a critical section then the matching loop tail (head) also occurs in some (possibly the same) segment comprising the same critical section.

In order to convert a cyclic CFG into an acyclic CFG we traverse the CFG CFG_i of thread T_i starting at its entry location in a depth-first manner and identify a set of back-edges. These back-edges transit from tails of loops to their matching heads. Deleting these back-edges results in an acyclic CFG which we denote by CFG'_i. Next we run Alg. 2 on CFG'_i, the only difference being that if during the backward traversal of a newly introduced locking statement we include a loop tail *lt* for the first time in the critical section induced by the newly introduced locking statements, then in order to preserve the consistency invariant we also need to include the matching loop head in the critical section (as was discussed in Sec. 3). Thus if *lh* is the matching loop head for *lt* and if *lh* does not already exists in the an atomic segment in the current specification then we generate a new instance of the lock insertion problem by inserting the atomic segment comprising the loop head *lh* in the existing set of atomic segments (steps 12-15 of Alg. 3).

Algorithm 2 Lock Insertion for General Programs

```
1: Input: Threads $T_1$ and $T_2$ specified it terms of their respective CFGs and pairs of segments $s_1 = [s_1^1, s_1^2], ...., s_k = [s_k^1, s_k^2]$, where $s_j^i$ is an atomic segment of $T_i$ specified as a DAG that is a subgraph of the CFG $CFG_i$ of $T_i$.
2: Assign a new lock $l_i$ to segment pair $s_i$.
3: Insert unlock statements $\mathsf{u}_1^i$ and $\mathsf{u}_2^i$ for lock $l_i$ in threads $T_1$ and $T_2$ immediately after the leaves of $s_i^1$ and $s_i^2$ in $T_1$ and $T_2$, respectively. (Insertion of Unlock Statements)
4: Insert lock statements $\mathsf{l}_1^i$ and $\mathsf{l}_2^i$ for lock $l_i$ in threads $T_1$ and $T_2$ immediately before the roots of $s_i^1$ and $s_i^2$ in $T_1$ and $T_2$, respectively.
5: for each lock $l_i$, where $i \in [1..k]$ do
6:   repeat
7:     Remove deadlocks involving $\mathsf{l}_i^1$ via Alg. 3
8:     Remove deadlocks involving $\mathsf{l}_i^2$ via Alg. 3
9:   until there do not exist any deadlocks involving $\mathsf{l}_i^1$ and $\mathsf{l}_i^2$
10: end for
```

VI. Implementation

From Alg. 1, we see that the key step in lock insertion involves deciding, in an efficient manner, the multiple reachability queries that are generated (via steps 6 and 9) as we traverse backwards along the CFGs of threads in the given program. However, in general, reachability of a pair of control locations in threads is not decidable. The strategy that is often used to bypass the decidability barrier is to consider reachability in the presence of synchronization primitives like locks and wait/notify only and ignore data variables. This is referred to as *static reachability*. We observe that relying on static reachability instead of reachability guarantees soundness of our procedure.

Algorithm 3 Remove Deadlocks

```
1: Input: Segments $s_{i1} = [s_{i1}^1, s_{i2}^2], ..., s_{ik_i} = [s_{ik_i}^1, s_{ik_i}^2]$, associated with the same lock $l_i$ and thread $T_m$, where $m \in [1..2]$.
2: Output: Possible re-location of lock(l) statements in thread $T_m$ in order to guarantee absence of deadlocks involving $\mathsf{l}_i^m$, i.e., the lock($l_i$) statement in $T_m$.
3: for each pair $s_{ij} = [s_{ij}^1, s_{ij}^2]$ do
4:   Set $Worklist$ to the locations of all the lock($l_i$) statements enforcing $s_{ij}^m$ in $T_m$ that are involved in a deadlock
5:   $Visited = \emptyset$
6:   while $Worklist \neq \emptyset$ do
7:     Remove a location loc from Worklist
8:     if there exists a lock $m$ held at loc that is acquired at a control location loc' in the segment $s_{ij}^{k'}$, where $k \neq k'$, and $(loc, loc')$ are pairwise reachable then
9:       for each predecessor $pred$ of $loc$ do
10:        if $pred \notin Visited$ then
11:          Add $pred$ to $Worklist$ and to $Visited$.
12:          if $pred$ is a loop tail lt that is not included in any of the segments $s_{ir}^m$, with $r \in [1..k_i]$ then
13:            Construct a new segment seg comprising only of the loop head lh that matches lt
14:            Add the new segment pair $sp = [sp^1, sp^2]$, where $sp^k = seg$ and $sp^{k'} = s_{ij}^{k'}$ with $k \neq k'$, and associate lock $\mathsf{l}_i$ with it
15:          end if
16:        end if
17:      end for
18:    else
19:      Insert lock($l_i$) immediately before $loc$
20:    end if
21:  end while
22: end for
```

A. Nested Locks

Alg. 1 formulates an optimal procedure for lock insertion for concurrent programs with arbitrary locking patterns. However, in real world applications most lock usage is *nested* [10], where we say that a concurrent program accesses locks in a *nested* fashion if along each computation of the program a thread can only release the last lock that it acquired along that computation and that has not yet been released. This has two main implications. First, it is known that while static reachability is undecidable for arbitrary locking patterns it not only becomes decidable for nested locks but efficiently so [10]. Thus a key advantage of nestedness is that it enable us to leverage efficient procedures for deciding static reachability thereby yielding a fast and effective lock insertion procedure.

Secondly, if a program has nested locks to start with, we would like to preserve this nestedness. Our general lock insertion procedure, however, may violate nestedness. We therefore formulate a modified procedure that ensures that nestedness is preserved in the newly synthesized program. Note that this procedure still guarantees optimality of newly introduced critical sections if we restrict ourselves to the space of programs with nested locks only.

B. Review of Acquisition Histories

We start by reviewing the notion of acquisition histories [10] that have been used for efficiently reasoning about static reachability for nested locks.

De nition (Acquisition History) *For a lock l held by thread T at a control location d, the acquisition history of l along a local computation x of T leading to c, denoted by $\mathsf{ah}_T(c, l, x)$, is the set of locks that have been acquired (and possibly released) by T since the last acquisition of l by T in traversing forward along x to c.*

Acquisition histories enable us to formulate a necessary and sufficient condition for static reachability for nested locks.

Theorem 2 (Decomposition Result) [9]. *Let x^i be a local computation of T_i leading to c_i. Then (c_1, c_2) is statically reachable via an interleaving of x^1 and x^2 if and only if (i) the locks held at c_1 and c_2 are disjoint, and (ii) the acquisition histories at c_1 and c_2 are consistent, i.e., there do not exist locks l and l' that are held at c_1 and c_2, respectively, such that $l \in \mathsf{ah}_{T_2}(c_2, l', x^2)$ and $l' \in \mathsf{ah}_{T_1}(c_1, l, x^1)$.*

The reason we refer to thm. 2 as the decomposition result is that it enables us to reason about static reachability for nested locks in a thread local manner. This is because much like locksets, acquisition histories can be computed *thread locally* at each location of interest in thread T via a simple traversal of the CFG of T. This is key to ensuring efficiency of deciding static reachability for nested locks.

Let $\mathsf{AH}_{T_i}(c_i)$ be the set of all possible acquisition histories encountered along paths of T_i leading to c_i. Then from thm. 2, we have the following criterion for static reachability between global control states.

Corollary (Generalized Decomposition Result). *Global control states $\mathsf{c} = (c_1, c_2)$ is statically reachable if and only if (1) disjoint sets of locks are held at c_1 and c_2, and, (ii) there exist acquisition histories $\mathsf{ah}_1 \in AH_{T_1}(c_1)$ and $\mathsf{ah}_2 \in AH_{T_2}(c_2)$ that are consistent.*

Nested Lock Insertion. In applying the decomposition result during lock insertion we face two main challenges. First, as we traverse the CFGs of threads backwards, we generate multiple (static) pairwise reachability queries. Thus we want to avoid computing acquisition histories between the same pair of control locations multiple times. Towards that end, we pre-compute, in one pre-processing step, the acquisition histories at all *relevant* control locations of interest in each thread.

Localizing the Analysis. The key issue next is how to localize these locations of interest. Towards that end, let LF_i be the set of last lock free (where no lock is held) locations along local paths of T_i leading to an entry location of the critical section to be enforced. Note that we can simply insert the *lock(l)* statement immediately after locations in LF_1 and LF_2. This ensures that desired critical sections are enforced and no new deadlocks are introduced as no lock is held at any of the locations in LF_1 or LF_2. However, this may not lead to optimal critical sections. It follows that, in order to achieve optimality, our lock insertion strategy can only introduce *lock(l)* statements immediately before an existing locking statement occurring along paths from locations in LF_i to the entry locations of the desired critical section. We call the set of all such locking statements *History Lock Statements* as they are in the acquisition history of the critical section that we are trying to enforce. Thus it suffices to compute $\mathsf{AH}_{T_i}(c_i)$ only for history lock statements c_i of T_i.

Once the acquisition histories have been computed, the procedure for lock insertion for the nested case can then be formulated as Alg. 4. Note that the main difference between algs. 1 and 4 is that pairwise reachability is determined using acquisition histories computed in steps 2-4. Alg. 2 for the general case can also be modified accordingly.

Algorithm 4 Nested Lock Insertion via Acquisition Histories

```
1:  Input: Threads $T_1$ and $T_2$ specified as control flow graphs, $CFG_1$ and $CFG_2$, respectively, and mutually atomic segments $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$ of $T_1$ and $T_2$, respectively.
2:  for each thread $T_i$ do
3:      Compute the lock acquisition histories $\mathsf{AH}_{T_i}(c)$ at each location $c$ where $c$ is a history lock statement of $C_i$ in $T_i$
4:  end for
5:  Insert $\mathsf{l}_1$ and $\mathsf{l}_2$ in threads $T_1$ and $T_2$ immediately before $c_i$ and $d_{i'}$, respectively.
6:  repeat
7:      if $\mathsf{l}_1$ can be involved in a potential deadlock then
8:          Move $\mathsf{l}_1$ backward along $T_1$ via a backward DFS traversal of $CFG_i$ till we reach control locations $c'$ of thread $T_1$ such that for each lock $m \in L_{[\mathsf{l}_2,\mathsf{u}_2]}$: either (i) $m$ is not held at $c'$, or (ii) for each location $d'$ in critical section $[\mathsf{l}_2, \mathsf{u}_2]$ where $m$ is acquired, $\mathsf{AH}_{T_i}(c')$ and $\mathsf{AH}_{T_i}(d')$ are not consistent.
9:      end if
10:     if $\mathsf{l}_2$ can be involved in a potential deadlock then
11:         Move $\mathsf{l}_2$ backward along $T_2$ till we reach a control location $d'$ of thread $T_2$ such that for each lock $m \in L_{[\mathsf{l}_1,\mathsf{u}_1]}$: either (i) $m$ is not held at $d'$, or (ii) for each location $c'$ in critical section $[\mathsf{l}_1, \mathsf{u}_1]$ where $m$ is acquired, $\mathsf{AH}_{T_1}(c')$ and $\mathsf{AH}_{T_2}(d')$ are not consistent.
12:     end if
13: until there do not exist any potential deadlocks involving $\mathsf{l}_1$ or $\mathsf{l}_2$
14: Add unlock statement to match $\mathsf{l}_1$ and $\mathsf{l}_2$ in a manner that ensure that locks are nested.
```

Example	KLOC	Segment Pairs	Acquisition History Computation	Lock Insertion (secs)
account.Main	50 LOC	2	0.9	0.1
atom001a	70 LOC	3	1.4	0.2
atom002a	75 LOC	3	1.6	0.3
banking-av	150 LOC	1	1.1	0.3
banking-sav	175 LOC	2	1.2	0.4
D-1	2.9	3	1.2	0.4
D-2	8.3	12	7.4	1.1
D-3	8.3	3	8	1.2
D-4	17.8	9	6.7	4.4
D-5	17.8	2	7	2.5

TABLE I
LOCK INSERTION DATA

C. Guaranteeing Nestedness of Locks

Alg. 4 does not guarantee preservation of nestedness of locks. To ensure nestedness we make two modifications to Alg. 4. First, instead of inserting the *unlock(l)* statement before the *lock(l)* statement, we first insert the *lock(l)* statement and then add the matching *unlock(l)* statements to ensure nestedness of locks. However, we need to make sure that the *lock(l)* statements are inserted at locations such that there exist locations where the matching *unlock(l)* statements can be inserted to ensure nestedness. This is accomplished by augmenting the conditions in steps 8 and 11 with the extra constraint that the matching unlock statements *unlock(l)* can be inserted so as to enforce the desired critical sections while preserving nestedness.

VII. EXPERIMENTS

We consider a set of public benchmarks with known atomicity violations used in our previous work [11]. These are small examples and are used mainly to illustrate the efficacy of our new lock insertion technique. We also use an in-house parallel implementation of an MPEG-4 decoder **S** with known atomicity violations detected via static and runtime techniques. Finally we also consider a large in-house concurrent software system implementing a distributed storage system, denoted by **D**. The **D** system consists of about 400K lines of C++ code using Boost libraries and is based on a thread pool model. We evaluated our approach by applying our technique to different modules of **D** denoted by **D**-1, **D**-2, **D**-3, **D**-4 and **D**-5.

We present the time taken for the context-sensitive points-to analysis for the lock pointers and the pre-processing step that computes the acquisition histories at locations of interest (col. 4) and the time taken for the lock insertion procedure (col. 5). The key thing worth noting is that the lock insertion procedure is efficient even for large examples (col. 5). In fact the total time taken is dominated by the points-to analysis and the acquisition history computation. This is to be expected as once the acquisition histories have been computed the lock insertion procedure involves highly localized dovetailed movements of the lock statements around critical sections to be enforced. Usually these movements are restricted to function boundaries. On the other hand, computing the points-to sets requires us to reason about code modules that may impact aliases of relevant lock pointers at locations of interest as opposed to just a few functions where the atomicity violations need to be fixed.

VIII. RELATED WORK AND CONCLUSION

There has been interesting work on automatically inferring locks for atomic sections [4], [12], [6], [1], [2]. However most of this work has focused on allocating/inferring locks for programs with no prior locks. The absence of locks allows one greater control over lock placement thereby making it easier to enforce the desired correctness properties. For instance, deadlocks can be prevented simply by allocating locks in a fixed global order. One does not have this freedom if locks are required to be inserted in a program with existing locks. This makes the problem of lock insertion more challenging than lock allocation/inference. There is also limited amount of work on exploiting program semantics to insert synchronization statements in order to fix bugs [5], enforce concurrency control in order to satisfy invariants [3], or ensure correctness [13]. However reasoning about program semantics requires the use of refined heavy-weight analyses like constraint/SAT solving or state space exploration via model checking.

In contrast, we have formulated a fully automatic, provably optimal, efficient and precise technique for lock insertion in concurrent code with pre-existing locks that ensures deadlock freedom while attempting to minimize the resulting critical sections. Importantly, our method localizes the analysis to only the necessary code modules. Moreover, for the special case of programs with nested locks our analysis is compositional, i.e., thread local, thereby avoiding a global analysis and ensuring scalability to large real-life programs.

REFERENCES

[1] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.
[2] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC*, 2008.
[3] Jyotirmoy V. Deshmukh, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In *ESOP*, 2010.
[4] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, 2007.
[5] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *SCOOL*, 2005.
[6] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *First Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.
[7] Takashi Horikawa. An approach for scalability-bottleneck solution: identification and elimination of scalability bottlenecks in a dbms. In *ICPE*, 2011.
[8] Takashi Horikawa. An approach for scalability-bottleneck solution: identification and elimination of scalability bottlenecks in a dbms (abstracts only). *SIGMETRICS Performance Evaluation Review*, 39(3), 2011.
[9] V. Kahlon and A. Gupta. On the Analysis of Interacting Pushdown Systems. In *POPL*, 2007.
[10] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.
[11] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.
[12] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.
[13] M. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.

Multi-Pushdown Systems with Budgets

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Othmane Rezine and Jari Stenman
Department of Information Technology
Uppsala University
Uppsala, Sweden

***Abstract*—We address the verification problem for concurrent programs modeled as multi-pushdown systems (MPDS). In general, MPDS are Turing powerful and hence come along with undecidability of all basic decision problems. Because of this, several subclasses of MPDS have been proposed and studied in the literature [1]–[4]. In this paper, we propose the class of bounded-budget MPDS where we restrict them in the sense that each stack can perform an unbounded number of context switches if its size is below a given bound, and is restricted to a finite number of context switches when its size is above that bound. We show that the reachability problem for this subclass is PSPACE-complete. Furthermore, we propose a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that running P under the bounded-budget restriction yields the same set of reachable states as running P'. By leveraging standard sequential analysis tools, we have implemented a prototype tool and applied it on a set of benchmarks, showing the feasibility of our translation.**

I. Introduction

In the last few years, a lot of effort has been devoted to the verification problem for models of concurrent programs (see, e.g., [1]–[3], [5]). On the other hand, pushdown systems have been proposed as an adequate formalism to describe sequential programs with procedure calls. Therefore, it is natural to model recursive concurrent programs as Multi-PushDown Systems (MPDS for short). However, MPDS are in general Turing powerful, and hence all the basic decision problems are undecidable for them. To overcome this barrier, several subclasses of multi-pushdown systems have been proposed and studied in the literature. The main goals of these works are (1) to explore the largest possible state space of the modeled concurrent program, and (2) to retain the decidability of some properties such as the reachability problem.

Context-bounding has been proposed in [1] as a suitable technique for the analysis of MPDS. The idea is to consider only runs of the system that can be divided into a given number of contexts, where in each context pop and push operations are exclusive to one stack. The state space which may be explored is still unbounded in the presence of recursive procedure calls, but the context-bounded reachability problem is NP-complete even in this case. Empirically, it has been shown that many concurrency errors, such as data races and atomicity violations, manifest themselves in executions with only a few contexts [6].

Another way to regain decidability is to consider depth-bounded verification for MPDS where the maximal possible depth (or size) of each stack is bounded by a given constant. In this case, the reachability problem becomes PSPACE-complete. However, since the explored state space is bounded, this approach is more suitable for detecting shallow bugs [7]. In fact, bounding the stack depth provides a completeness result for the case where the threads are modeled as finite-state systems (this is not the case for the context-bounded analysis).

This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

In this paper, we generalize both context-bounded analysis and depth-bounded verification by introducing the class of MPDS *with budgets*. Intuitively, for each thread (or stack), we associate two values $k, d \in \mathbb{N} \cup \{\omega\}$ (where ω is the first limit ordinal) such that each thread can perform at most k consecutive context switches unless its stack depth goes below the given bound d. More precisely, each thread is given a budget b of contexts. The thread then operates in two modes, I and II. In mode I, the stack depth of the thread is less than or equal to d, while in mode II it is strictly above d. The budget of the thread is unbounded in mode I, i.e., $b = \infty$. In other words, the thread is allowed to perform any number of context switches while it is in mode I. As soon as the stack depth of the thread grows above d, the thread enters mode II and its budget b is set to k. Each time the thread performs a context switch in mode II, its budget b is decremented by one. The thread leaves mode II in one of two ways: either it consumes all its budget (its budget b becomes negative) in which case the thread will be blocked; or the stack depth of the thread becomes d in which case it enters mode I and its budget is reset to unbounded ($b = \omega$) again.

We identify two subclasses of MPDS with budgets. We call the first subclass *uniformly bounded-budget MPDS*. Here, we associate finite values to the stack depth $d \in \mathbb{N}$ and context budget $k \in \mathbb{N}$ for each thread (or stack). For this case, we show that the reachability problem is PSPACE-complete. The lower bound is proved by a straightforward reduction from the non-emptiness test of the intersection of a finite set of regular languages (which is PSPACE-complete). To prove the upper-bound, we show that it is possible to reduce, in polynomial time, the reachability problem for a uniformly bounded-budget MPDS to the non-emptiness test for the synchronous product of a finite set of depth-bounded pushdown automata (which is PSPACE-complete).

Then, we consider the class of *singly unbounded-budget* MPDS where we have at most one thread that can perform an unbounded number of context switches regardless of its stack depth, and all the other threads have finite values for their stack depth and context-budget bounds. We show that the reachability problem for this class is EXPTIME-complete. The lower bound is proved by a reduction from the non-emptiness test of the intersection of a pushdown automaton with several regular languages (which is EXPTIME-complete). For the upper bound, we show that the reachability problem for a singly unbounded-budget MPDS can be reduced to the emptiness problem for a pushdown automaton whose size is exponential.

In the second part of the paper, we investigate the issue of defining a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that running P under the uniformly bounded-budget restriction yields the same set of reachable states as running P'. In other words, we have reduced the problem of verifying (an under-approximation of) the concurrent P to that of verifying a sequential program P'. In fact, the only source of abstraction in our translation is the fact that we limit the behavior of P when its stack depth exceeds the given limit. In particular, our translation preserves the data domains of the original program in the sense that P and P' have the same features (e.g., recursive procedure calls, type of data structures). We show that the translation can be performed using additional copies of the shared variables and local variables. More importantly, the fact that P' is a sequential program means that our translation allows us to use existing analysis and verification tools designed for sequential programs in order to perform the same kind of analysis and verification for concurrent programs under uniformly bounded-budget restriction. To show its use in practice, we have implemented our approach and applied it on several examples, using the three back end tools MOPED [8], ESBMC [9], and CBMC [7]. We also compare our results to the ones obtained using concurrent verification tools, namely ESBMC [9] and POIROT [10]. In our experiments, bugs (i.e. violations of state invariants) appear for small bounds.

Related work: Our model is inspired by the work of Finkel and Sangnier [11], where they propose an extension of reversal-bounded counter machines, restricting each counter to a finite number of alternations between the increasing and decreasing modes when its value goes beyond a given bound.

As mentioned earlier, several decidable classes of multi-pushdown systems have been proposed [1]–[3], [5]. The closest model to multi-pushdown systems with budgets is Scope-bounded Multistack PushDown Systems (SMPDS for short) [5] where each symbol in a stack can be popped only if it has been pushed within a bounded number of context switches. We can show that the reachability problem for SMPDS can be reduced to the corresponding one for uniformly bounded-budget MPDS where the value of the stack depth is 0. This can be done by assuming that a stack symbol that will never be popped in the context of [5] will not be pushed into the stack. Thus, each symbol that is pushed into the stack should be removed within k context-switches. On the other hand, simulating uniformly bounded-budget MPDS by SMPDS does not seem to be straightforward without an exponential explosion (to encode the content of each stack up to the stack depth bound).

To the best of our knowledge there is no decidable subclass of multi-stack pushdown system similar to the class of singly unbounded-budget MPDS (for which we show the reachability problem to be EXPTIME-COMPLETE).

Our code-to-code translation follows the line of research on compositional reductions from concurrent to sequential programs [12]–[15]. Recently, La Torre and Parlato have proposed in [16] a sequentialization for SMPDS where for each SMPDS, they construct an equivalent single-stack pushdown system that faithfully simulates the behavior of each thread. However, the proposed sequentialization has not been implemented and so we were not able to compare it with our translation. Moreover, the two translation schemes were developed independently and simultaneously [17].

II. PRELIMINARIES

In this section, we fix some basic definitions and notations that will be used in the rest of the paper. We assume that the reader is familiar with automata and language theory.

a) Notations: Let $\mathbb{N}$ denote the non-negative integers, and let $\mathbb{N}^k$ and $\mathbb{N}^k_\omega$ denote the set of vectors of dimension k over $\mathbb{N}$ and $\mathbb{N} \cup \{\omega\}$, respectively ($\omega$ representing the first limit ordinal). For every $i, j \in \mathbb{N}_\omega$ such that $i \leq j$, we use $[i..j]$ to denote the set $\{k \in \mathbb{N}_\omega \mid i \leq k \leq j\}$.

Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^+) the set of all words (resp. non empty words) over Σ, and by ϵ the empty word. A language is a (possibly infinite) set of words. Let u be a word over Σ. The length of u is denoted by $|u|$ (we have $|\epsilon| = 0$).

Let L be a language over Σ and let $w \in \Sigma^*$ be a word. We define $w.L = \{w.u \mid u \in L\}$. We define the *shuffle* operator $\sqcup\!\sqcup$ over two words inductively as $\sqcup\!\sqcup(\epsilon, w) = \sqcup\!\sqcup(w, \epsilon) = \{w\}$ and $\sqcup\!\sqcup(a.u^{'}, b.v^{'}) = a.(\sqcup\!\sqcup(u^{'}, b.v^{'}) \cup b.(\sqcup\!\sqcup(a.u^{'}, v^{'})$. Given two languages L_1 and L_2, we define their shuffle as $\sqcup\!\sqcup(L_1, L_2) = \bigcup_{u \in L_1, v \in L_2} \sqcup\!\sqcup(u, v)$. The shuffle operator for multiple languages can be extended analogously.

b) Pushdown Automata: A pushdown automaton is defined by a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where: (1) Q is a finite non-empty set of states, (2) Σ is the input alphabet, (3) Γ is the stack alphabet, (4) Δ is the finite set of transition rules of the form $(q, u) \xrightarrow{a} (q', u')$ where $q, q' \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $u, u' \in \Gamma^*$ such that $|u| + |u'| \leq 1$, (5) $I \subseteq Q$ is the set of initial states, and (6) $F \subseteq Q$ is the set of final states. The size of $\mathcal{P}$ is defined by $|\mathcal{P}| = |Q| + |\Sigma| + |\Gamma|$.

A *configuration* of $\mathcal{P}$ is a tuple (q, σ, w) where $q \in Q$ is the current state, $\sigma \in \Sigma^*$ is the remaining input word, and $w \in \Gamma^*$ is the stack content. We define the binary relation $\Rightarrow_\mathcal{P}$ between configurations as follows: $(q, a\sigma, uw) \Rightarrow_\mathcal{P} (q', \sigma, u'w)$ iff $(q, u) \xrightarrow{a} (q', u')$. The transition relation $\Rightarrow^*_\mathcal{P}$ is the reflexive transitive closure of $\Rightarrow_\mathcal{P}$.

The language $L(\mathcal{P})$ accepted by $\mathcal{P}$ is defined by the set of finite words $\sigma \in \Sigma^*$ such that $(q_{\mathsf{init}}, \sigma, \epsilon) \Rightarrow^*_\mathcal{P} (q_{\mathsf{final}}, \epsilon, \epsilon)$ for some $q_{\mathsf{init}} \in I$ and $q_{\mathsf{final}} \in F$.

Let $d \in \mathbb{N}$. We define the transition relation $\rightarrow_{>d}$ between configurations of $\mathcal{P}$ as follows: $(q, \sigma, w) \rightarrow_{>d} (q', \sigma', w')$ if and only if $(q, \sigma, w) \Rightarrow_{\mathcal{P}} (q', \sigma', w')$ and $|w'| > d$ or $|w| > d$. Intuitively, the transition relation $\rightarrow_{>d}$ can be performed only if the stack depth of the starting or target configuration is at least $d + 1$. The transition relation $\rightarrow^*_{>d}$ is the reflexive transitive closure of $\rightarrow_{>d}$.

Similarly, we can define the transition relation $\rightarrow_{\leq d}$ between configurations of $\mathcal{P}$ as follows: $(q, \sigma, w) \rightarrow_{\leq d} (q', \sigma', w')$ if and only if $(q, \sigma, w) \Rightarrow_{\mathcal{P}} (q', \sigma', w')$, $|w'| \leq d$ and $|w| \leq d$. Intuitively, the transition relation $\rightarrow_{\leq d}$ can only be performed when the stack depths of both the starting and target configurations are at most d. The transition relation $\rightarrow^*_{\leq d}$ is the reflexive transitive closure of $\rightarrow_{\leq d}$.

Given $d, k \in \mathbb{N}$, we define the relation $\rightarrow_{(k,d)}$ between configurations of depth d as follows: $(q, \sigma, w) \rightarrow_{(k,d)} (q', \sigma', w')$ if and only if $(q, \sigma, w) \rightarrow^*_{>d} (q', \sigma', w')$, $|\sigma| - |\sigma'| \leq k$, $w' = w$, and $|w| = d$. This means that the pushdown automaton can only read k consecutive input symbols without its stack depth going below the bound d.

Let $L_{(k,d)}(\mathcal{P})$ denote the set of words $\sigma \in \Sigma^*$ such that there is a sequence of configurations $c_0, c_1, \ldots, c_n$ where (1) c_0 is of the form (q_0, σ, ϵ) with $q_0 \in I$, (2) c_n is of the form $(q_n, \epsilon, \epsilon)$ with $q_n \in F$, and (3) for every $i \in [1..n]$, we have $c_{i-1} \rightarrow_{(k,d)} c_i$ or $c_{i-1} \rightarrow^*_{\leq d} c_i$ holds. We call $L_{(k,d)}(\mathcal{P})$ the (k, d)-bounded language of $\mathcal{P}$.

We also define the language $L_{(-1,d)}(\mathcal{P})$ to be the set of words $\sigma \in \Sigma^*$ such that $(q_{\mathsf{init}}, \sigma, \epsilon) \rightarrow^*_{\leq d} (q_{\mathsf{final}}, \epsilon, \epsilon)$ where $q_{\mathsf{init}} \in I$ and $q_{\mathsf{final}} \in F$. Intuitively, the set $L_{(-1,d)}(\mathcal{P})$ (or simply $L_d(\mathcal{P})$ when it is clear from the context) contains all words accepted by the runs of $\mathcal{P}$ where the stack depth is always bounded by d.

Lemma 1: Let $d, k \in \mathbb{N}$ and $\mathcal{P}$ be a pushdown automaton. Then, it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$.

Proof: To prove this, it is sufficient to show the following lemma:

Lemma 2: Let $k \in \mathbb{N}$ be a natural number and $\mathcal{P}$ be a pushdown automaton. Then, it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_k(\mathcal{P}') = L(\mathcal{P}) \cap \Sigma^{\leq k}$.

Proof: Let us first recall some basic results about context-free languages.

A *context-free grammar* (CFG) G is a tuple $(\mathcal{X}, \Sigma, R, S)$ where $\mathcal{X}$ is a finite non-empty set of *variables* (or *nonterminals*), Σ is an alphabet of *terminals*, $R \subseteq \big(\mathcal{X} \times (\mathcal{X}^2 \cup \Sigma)\big) \cup (S \times \{\epsilon\})$ a finite set of *productions* (the production (X, w) may also be denoted by $X \rightarrow w$), and $S \in \mathcal{X}$ is a start variable. The size of G is defined by $|G| = (|\mathcal{X}| + |\Sigma|)$. Observe that the form of the productions is restricted, but it has been shown in [4] that every CFG can be transformed, in polynomial time, into an equivalent grammar of this form.

Given strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we say $u \Rightarrow_G v$ if there exists a production $(X, w) \in R$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Rightarrow^*_G$ for the reflexive transitive closure of $\Rightarrow_G$. We define the context-free language generated by $L(G)$ as $\{w \in \Sigma^* \mid S \Rightarrow^*_G w\}$.

Let $k \in \mathbb{N}$. A derivation α given by $\alpha \overset{\text{def}}{=} \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \cdots \Rightarrow_G \alpha_n$ is *k-bounded* if $|\alpha_i| \leq k$ for all $i \in [1..n]$. We denote by $L^{(k)}(G)$ the subset of $L(G)$ such that for every $w \in L^{(k)}(G)$ there exists a k-bounded derivation $S \Rightarrow^*_G w$. We call $L^{(k)}(G)$ the *k-bounded approximation* of $L(G)$.

Lemma 3: Given a context-free grammar G and $k \in \mathbb{N}$, then it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}$ such that $L_k(\mathcal{P}) = L^{(k)}(G)$.

Proof: Since the context-free grammar G is in the normal form, we know that any k-bounded derivation have a derivation tree $\mathcal{T}$ in which the number of leaves is at most k. Moreover, any path of the derivation tree $\mathcal{T}$ has at most k nodes and each node has at most two outgoing edges. This implies that the we can construct a stateless pushdown automaton $\mathcal{P}'$ whose alphabet is exactly the set of variables of G. The derivation tree of the grammar G is simulated by the pushdown automaton $\mathcal{P}'$ in a leftmost way in the standard manner. Then, we construct a pushdown automaton $\mathcal{P}$ which results from the intersection of the pushdown automaton $\mathcal{P}'$ and the finite state automaton that recognizes words of length at most k. Now, it is easy to see that $L(\mathcal{P}) = L_k(\mathcal{P}) = L^{(k)}(G)$. ■

To prove lemma 2, we will make use of the fact that for every pushdown automaton $\mathcal{P}$, it is possible to construct, in polynomial time in the size of $\mathcal{P}$, a context-free grammar G such that $L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$ [18]. Moreover, we can assume that this context-free grammar is in the normal form (based on the result in [4] showing that every context-free grammar can be transformed, in polynomial time, into an equivalent grammar in the normal form). Then, we can apply Lemma 3 to construct, in polynomial time, the pushdown automaton $\mathcal{P}'$ such that $L_k(\mathcal{P}) = L^{(k)}(G)$. Hence, we have $L_k(\mathcal{P}) = L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$. ■

Let $\mathcal{P}$ be a pushdown automaton. To prove Lemma 1, we construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$ as follows: The pushdown automaton $\mathcal{P}'$ mimics the pushdown automaton $\mathcal{P}$ if the current stack depth of $\mathcal{P}$ is less or equal to d. Moreover, $\mathcal{P}'$ keeps track of the current stack depth in its control state. If the current depth of the stack of $\mathcal{P}$ (and therefore $\mathcal{P}'$) is precisely d and $\mathcal{P}$ performs a push transition t from a state q, then the pushdown automaton $\mathcal{P}'$ guesses the return state q' (when the stack of $\mathcal{P}$ is again of depth d). Then, $\mathcal{P}'$ starts to mimic the pushdown automaton $\mathcal{P}''$ (constructed using Lemma 2) from the pushdown automaton $\mathcal{P}'''$ built from $\mathcal{P}$ by setting the initial state of $\mathcal{P}'''$ to q and the final state of $\mathcal{P}'''$ to q'. Moreover, we constrain the pushdown automaton $\mathcal{P}'''$ such that the only first possible simulated transition of $\mathcal{P}$ is precisely the push transition t and the pushdown automaton $\mathcal{P}'''$ halts when its stack is empty (except for the initial configuration). To detect that the stack of $\mathcal{P}''$ is empty, we can use a special symbol $\perp$ to mark the bottom of the stack. By construction, the pushdown automaton $\mathcal{P}''$ accepts exactly the words of length less than k (which are the set of words σ generated by the run of $\mathcal{P}$ of the form $(q, \sigma, \epsilon) \rightarrow_{(k,d)} (q', \epsilon, \epsilon)$ and where $|\sigma| \leq k$). Observe that $(q, \sigma\sigma', w) \rightarrow_{(k,d)} (q', \sigma, w)$ for some w such that $|w| = d$ holds if and only if $(q, \sigma, \epsilon) \rightarrow_{(k,d)} (q', \epsilon, \epsilon)$ holds. Since the stack depth of $\mathcal{P}''$ is at most k, the stack depth of $\mathcal{P}$ is at most

$d + k$. ■

Given pushdown automata $\mathcal{P}_0, \ldots, \mathcal{P}_n$ and bounds $d_0, \ldots, d_n \in \mathbb{N}$, we define *the non-emptiness test of the synchronization of depth-bounded pushdown automata* as the problem of checking the emptiness of the language $L_{d_0}(\mathcal{P}) \cap \sqcup\!\sqcup(L_{d_1}(\mathcal{P}_1), \ldots, L_{d_n}(\mathcal{P}_n))$.

Lemma 4: The non-emptiness test of the synchronization of depth-bounded pushdown automata is PSPACE-complete.

Proof: The upper-bound can be obtained by an easy reduction to the emptiness problem for a Turing machine having $n+1$-tapes, where each tape $i \in [0..n]$ has d_i cells.

The lower bound follows by a reduction from the non-emptiness test of the intersection of several regular languages (particular case of depth-bounded pushdown automata) which is known to be PSPACE-hard. ■

III. Multi-Pushdown Systems

In this section, we recall the definition of *multi-pushdown systems*. Multi-pushdown systems (or *MPDS* for short) have a finite set of states along with a finite number of read-write memory tapes (stacks) with a last-in-first-out rewriting policy. The types of transitions that can be performed by a MPDS are: (i) pushing a symbol into one stack, (ii) popping a symbol from one stack, or (iii) an internal action that changes the state of the automaton while keeping the stacks unchanged. Note that since we are not interested in this model as a language acceptor, it does not include an input alphabet or final states.

Definition 1: A *multi-pushdown system* (MPDS) is a tuple $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ where $n \geq 1$ is the number of stacks, Q is a finite set of *states*, Γ is the *stack alphabet*, $\Delta \subseteq \big(Q \times [1..n] \times Q\big) \cup \big(Q \times [1..n] \times Q \times \Gamma\big) \cup \big(Q \times \Gamma \times [1..n] \times Q\big)$ is the *transition relation*, and q_{init} is the *initial state*.

Let $q, q' \in Q$ be two states, $i \in [1..n]$ a stack index, and $\gamma \in \Gamma$ a stack symbol. A transition of the form (q, i, q') is an internal operation that moves the state from q to q' while keeping the contents of the stacks unchanged. The stack index i is included in this operation for technical reasons. A transition of the form (q, i, q', γ) corresponds to a push operation that changes the state from q to q', and adds the symbol γ to the top of the i-th stack. Finally, a transition of the form (q, γ, i, q') corresponds to a pop operation that moves the state from q to q', and removes the symbol γ from the top of the i-th stack.

A configuration c of $\mathcal{M}$ is an $(n+1)$-tuple $(q, w_1, \ldots, w_n)$ where $q \in Q$ is a state and for every $i \in [1..n]$, $w_i \in \Gamma^*$ is the content of the i-th stack. We use $State(c)$ and $Stack_i(c)$, with $i \in [1..n]$, to respectively denote q and w_i. We denote by $c^{\mathsf{init}}_{\mathcal{M}} = (q_{\mathsf{init}}, \epsilon, \epsilon, \ldots, \epsilon)$ the initial configuration of $\mathcal{M}$.

We define the transition relation $\rightarrow_{\mathcal{M}}$ on the set of configurations as follows. For configurations $c = (q, w_1, \ldots, w_n)$ and $c' = (q', w'_1, \ldots, w'_n)$, an index $i \in [1..n]$, and a transition $t \in \Delta$, we write $c \xrightarrow{t}_{\mathcal{M}} c'$ to denote that one of the following cases holds:

- **Internal operation:** $t = (q, i, q')$ and $w'_j = w_j$ for all $j \in [1..n]$.
- **Push operation:** $t = (q, i, q', \gamma)$ for some $\gamma \in \Gamma$, $w'_i = \gamma \cdot w_i$, and $w'_j = w_j$ for all $j \in ([1..n] \setminus \{i\})$.
- **Pop operation:** $t = (q, \gamma, i, q')$ for some $\gamma \in \Gamma$, $w_i = \gamma \cdot w'_i$, and $w'_j = w_j$ for all $j \in ([1..n] \setminus \{i\})$.

A *computation* π of $\mathcal{M}$ from a configuration c to a configuration c' is a sequence $c_0 t_1 c_1 t_2 \cdots t_m c_m$ such that: (1) $c_0 = c$ and $c_m = c'$, and (2) $c_{i-1} \xrightarrow{t_i}_{\mathcal{M}} c_i$ for all $i \in [1..m]$; each configuration c_i is said to be *reachable* from c. We use $initial(\pi)$ and $target(\pi)$ to denote respectively c_0 and c_m.

Given two computations $\pi_1 = c_0 t_1 \cdots t_m c_m$ and $\pi_2 = c_{m+1} t_{m+2} \cdots t_k c_k$, π_1 and π_2 are said to be *compatible* if $c_m = c_{m+1}$. Then, we write $\pi_1 \bullet \pi_2$ to denote the computation $\pi \overset{\text{def}}{=} c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m t_{m+2} c_{m+2} t_{m+3} \cdots\cdots t_k c_k$.

In the following, we propose the class of *bounded-budget* computations of MPDS. Intuitively, with each stack $i \in [1..n]$, we associate two values $k_i, d_i \in \mathbb{N}_\omega$ such that the stack i can perform at most k_i contexts without its size going below d_i. A context is a run of $\mathcal{M}$ where operations are exclusive to one stack. (Observe that k_i and d_i could be ω.) Next, we describe bounded-budget computations formally.

Contexts: A *context* of a stack $i \in [1..n]$ is a computation of the form $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$ in which $t_j \in \Delta_i \overset{\text{def}}{=} \big(Q \times \{i\} \times Q\big) \cup \big(Q \times \{i\} \times Q \times \Gamma\big) \cup \big(Q \times \{i\} \times \Gamma \times Q\big)$ for all $j \in [1..m]$. Observe that every computation can be seen as the concatenation of a sequence of contexts $\pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_\ell$.

For any two contexts π_1 and π_2 of the stack i, we write $\pi_1 \bullet_i \pi_2$ to denote that $Stack_i(initial(\pi_2)) = Stack_i(target(\pi_1))$ (i.e., in this case we say that π_1 and π_2 are compatible w.r.t. stack i). This notation is extended in a straightforward manner to sequence of contexts. Observe that if $\pi = \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_m$ is a computation where each π_j is a context, then if $i_1 < i_2 < \ldots < i_k$ are all the indices j such that π_j is a context of stack i, then $\pi_{i_1} \bullet_i \pi_{i_2} \bullet_i \ldots \bullet_i \pi_{i_k}$.

A context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$ of the stack $i \in [1..n]$ is said to be of depth at most (resp. least) $d \in \mathbb{N}$ if and only if for every $j \in [0..m]$, $|Stack_i(c_j)| \leq d$ (resp. $|Stack_i(c_j)| \geq d$). The definition is extended in the straightforward manner to sequences of contexts as follows: The sequence $\pi = \pi_1 \bullet_i \pi_2 \bullet_i \ldots \bullet_i \pi_m$ of compatible contexts of the stack i is of depth at most (resp. least) $d \in \mathbb{N}$ iff for every $j \in [1..m]$, π_j is of depth at most (resp. least) d.

Block: A *block* ρ of a stack $i \in [1..n]$ of size $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$ is a sequence of compatible contexts of the form $c_0 t_0 \cdot \pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_m \cdot t_m c_m$ of stack i such that $|Stack_i(c_0)| = |Stack_i(c_m)| = d$ and π_j is a context of depth at least $d+1$ for all $j \in [1..m]$.

Budget-Bounded Computations: Intuitively, in a budget-bounded computation, we associate with each stack $i \in [1..n]$, a budget of contexts $k_i \in \mathbb{N}_\omega$ and depth bound $d_i \in \mathbb{N}_\omega$ such that if we consider a point in the computation where the stack i is of depth d_i and a symbol is being pushed into this stack (i.e., the depth of the stack now becomes $d_i + 1$), then this newly pushed stack symbol should be removed within k_i contexts involving this stack i. This implies that, in a budget-bounded computation, each computation of the stack i is a concatenation of contexts of depth at most d_i and blocks of size k_i and depth d_i. The formal definition is as follows:

Let π be a computation of $\mathcal{M}$. Let $\bar{k} = (k_1, k_2 \ldots, k_n) \in \mathbb{N}_\omega^n$ be the context-budget vector and $\bar{d} = (d_1, d_2, \ldots, d_n) \in \mathbb{N}_\omega^n$ the stack depth vector. We say that π is $(\bar{k}, \bar{d})$*-budget-bounded* if it can be written as a concatenation $\pi_1 \bullet \pi_2 \bullet \cdots \bullet \pi_m$ of contexts (observe that for all j, π_j and π_{j+1} could be contexts of the same stack) in such a way that if $\sigma_i = \pi_{i_1}^i \bullet_i \pi_{i_2}^i \bullet_i \cdots \bullet_i \pi_{m_i}^i$ (with $i_1 < i_2 < \cdots < m_i$) is the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$, then there is a sequence $\rho_i = \rho_1^i \bullet_i \rho_2^i \bullet_i \cdots \bullet_i \rho_{\ell_i}^i$ of contexts of depth at most d_i and blocks of size k_i and depth d_i such that $\sigma_i = \rho_i$.

By restricting the allowed bound vectors $(\bar{k}, \bar{d})$, we can distinguish two sub-classes of MPDS under budget-bounding.

Definition 2: A $(\bar{k}, \bar{d})$-budget-bounded computation π is a *singly unbounded-budget* computation if and only if there is at most one index $i \in [1..n]$ such that either $k_i = \omega$ or $d_i = \omega$.

In singly unbounded-budget computations, we have at most one stack $i \in [1..n]$ that can perform an unbounded number of contexts regardless of its depth. Any other stack j (with $i \neq j$) of $\mathcal{M}$ can at most perform a *finite* number consecutive contexts without its size going below a given *finite* bound.

Definition 3: A $(\bar{k}, \bar{d})$-budget-bounded computation π is a *uniformly bounded-budget* computation if and only if for every $i \in [1..n]$, we have $k_i \in \mathbb{N}$ and $d_i \in \mathbb{N}$.

Observe that in the case of uniformly bounded-budget computations, each stack $i \in [1..n]$ has a *finite* context-budget and a *finite* stack depth bound.

IV. The Budget-Bounded Reachability Problem

In this section, we study the decidability and complexity of the reachability problem for MPDS under budget-bounding. Let $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ be a MPDS. Let $\bar{k} = (k_1, k_2 \ldots, k_n) \in \mathbb{N}_\omega^n$ be the context-budget vector and $\bar{d} = (d_1, d_2, \ldots, d_n) \in \mathbb{N}_\omega^n$ the stack depth vector. The $(\bar{k}, \bar{d})$*-budget-bounded reachability problem* is to determine, for a given state $q_{\mathsf{final}} \in Q$, whether there is a $(\bar{k}, \bar{d})$-budget-bounded computation from the initial configuration $c_{\mathcal{M}}^{\mathsf{init}}$ to the configuration $(q_{\mathsf{final}}, \epsilon, \ldots, \epsilon)$. The input size of this problem is $n + |Q| + |\Gamma| + |\Delta| + k + d$, where k and d are the largest natural numbers (or 0 if they do not exist) in the vectors $\bar{k}$ and $\bar{d}$, respectively.

A. The Uniformly Budget-Bounded Reachability Problem

In the following, we show that the reachability problem for MPDS restricted to only uniformly budget-bounded computations is PSPACE-complete.

Theorem 5: The $(\bar{k}, \bar{d})$-budget-bounded reachability problem for MPDS is PSPACE-complete if for every $i \in [1..n]$, we have $k_i \in \mathbb{N}$ and $d_i \in \mathbb{N}$.

The rest of this section is devoted to the proof of Theorem 5. The lower bound follows by a reduction from the non-emptiness test of the intersection of several regular languages (which is known to be PSPACE-hard).

To prove the upper bound, we reduce the reachability problem for MPDS restricted to only uniformly budget-bounded computations to the non-emptiness test of the synchronization of depth-bounded pushdown automata which is PSPACE-complete (see Lemma 4). The idea behind the proof is the following: Let ρ be a $(\bar{k}, \bar{d})$-uniformly budget-bounded computation and let $i \in [1..n]$ be a stack of $\mathcal{M}$. Then, we know that the projection of π on the set of transitions performed by stack i is a compatible sequence ρ_i of contexts of the form $\pi_1^i \bullet_i \pi_2^i \bullet_i \cdots \bullet_i \pi_{m_i}^i$. Since the communication between stacks is done via control states, we can summarize each context π_j^i (with $j \in [1..m_i]$) by a pair of states of the form $(q_j^i, {q'}_j^i)$ where q_j^i (resp. ${q'}_j^i$) is the state at the beginning (resp. end) of the context π_j. Then, we can summarize the stack computation ρ_i by the summary sequence $(q_1^i, {q'}_1^i)(q_2^i, {q'}_2^i) \cdots (q_{m_i}^i, {q'}_{m_i}^i)$. We show that it is possible to compute a pushdown automaton $\mathcal{P}_i$ such that the set of all possible summary sequences that can be generated by stack i along a $(\bar{k}, \bar{d})$-uniformly budget-bounded computation can be characterized by the $(-1, d+k)$-bounded language of $\mathcal{P}_i$. Then, we show that we can put together all summary traces and hence produce only consistent interleavings of these summaries (for all stacks) that arises from $(\bar{k}, \bar{d})$-uniformly budget-bounded computation.

Before we present the details, we introduce some notations and definitions that will be useful. For any context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$, we can associate a tuple $\mathit{Summary}(\pi) = (q, q')$ of the pair of states encountered at the beginning and end of the context π (i.e., $q = \mathit{State}(c_0)$ and $q' = \mathit{State}(c_m)$). Let $\rho = \pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_\ell$ be a sequence of contexts for some $i \in [1..n]$. We can then extend the definition of context summaries to sequence of contexts as follows: $\mathit{Summary}(\rho) = \mathit{Summary}(\pi_1)\mathit{Summary}(\pi_2) \cdots \mathit{Summary}(\pi_\ell)$. The function *Summary* is also extended in straightforward manner to blocks and sequences of blocks and contexts.

Let $w = (q_1, q_1')(q_2, q_2') \cdots (q_m, q_m')$ be a word over the summary alphabet $Q \times Q$. The word (or summary) w is said to be *consistent* if $q_1 = q_{\mathsf{init}}$, $q_m' = q_{\mathsf{final}}$ and $q_j' = q_{j+1}$ for all $j \in [1..m-1]$. Observe that the set of all consistent summaries can be recognized by a finite state automaton (i.e., a pushdown automaton of depth 0) whose size is polynomial in $\mathcal{M}$. Let $\mathcal{P}_0$ be such a pushdown automaton.

Let π be a $(\bar{k}, \bar{d})$-budget-bounded computation that reaches the state q_{final}. We can assume that π is of the form $\pi_1 \bullet \pi_2 \bullet \pi_3 \bullet \cdots \bullet \pi_m$ where each π_j, with $j \in [1..m]$, is a stack context. Then, let $\sigma_i = \pi_{i_1}^i \bullet_i \pi_{i_2}^i \bullet_i \pi_{i_3}^i \bullet_i \cdots \bullet_i \pi_{m_i}^i$ (with $i_1 < i_2 < i_3 < \cdots < m_i$) be the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$. By definition, we know that for any stack $i \in [1..n]$, there is a sequence $\rho_i = \rho_1^i \bullet_i \rho_2^i \bullet_i \cdots \bullet_i \rho_{\ell_i}^i$ of contexts of depth at most d_i and blocks of size k_i and depth d_i such that $\sigma_i = \rho_i$.

Then, it is easy to see that there is a consistent word in $\sqcup\!\sqcup(\{\mathit{Summary}(\sigma_1)\}, \ldots, \{\mathit{Summary}(\sigma_n)\})$. On the other hand, we can show that if for every stack $i \in [1..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$, then M has a $(\bar{k},\bar{d})$-budget-bounded computation that reaches q_{final}.

Now, we can show that checking the existence of such a consistent word can be reduced, in polynomial time, to the non-emptiness test of the synchronization of depth-bounded pushdown automata (which is PSPACE-complete), and hence we obtain the completeness of Theorem 5.

Lemma 6: The problem of checking whether for every $i \in [1..n]$ there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for the stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ can be reduced to the non-emptiness test of the synchronization of depth-bounded pushdown automata.

Proof (sketch). For every $i \in [1..n]$, let $L_i(\mathcal{M})$ be the set of words $Summary(\sigma_i)$ where σ_i is a compatible sequence of contexts of depth at most d_i and blocks of size k_i and depth d_i for stack i. Then, we can construct if $k_i > 0$ (resp. $k_i = 0$), in polynomial time, a pushdown automaton $\mathcal{P}_i$ whose (k_i, d_i)-bounded (resp. $(-1, d_i)$-bounded) language is precisely $L_i(\mathcal{M})$. The pushdown automaton $\mathcal{P}_i$ performs the same operations on its state and stack as the ones specified by Δ_i (i.e., the set of operations of stack i). More precisely, $\mathcal{P}_i$ (1) guesses the occurrence of a context π_i of stack i while making visible as a transition label its summary $Summary(\pi_i) = (q_i, q'_i)$, and (2) checks if from the current stack content and the state q_i, the state q'_i is reachable (and this will mark the end of the simulation of the context π_i). Moreover, $\mathcal{P}_i$ guesses for each context if it is a context of depth at most d_i or a context belonging to a block of size k_i and depth d_i (in the latter case $\mathcal{P}_i$ guesses also its position inside the block), then checks that all these assumptions hold when checking the feasibility of such contexts.

Now, we can apply Lemma 1 to construct, for each pushdown automaton $\mathcal{P}_i$, a bounded-depth pushdown automaton $\mathcal{P}'_i$ such that $L_{k_i+d_i}(\mathcal{P}'_i) = L_i(\mathcal{M})$. Then, checking whether for every $i \in [1..n]$ there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ boils down to checking the non-emptiness of the language $L_0(\mathcal{P}_0) \cap \sqcup\!\sqcup(L_{k_1+d_1}(\mathcal{P}'_1),\ldots,L_{k_n+d_n}(\mathcal{P}'_n))$.

B. The Singly Unbounded-Budget Reachability Problem

In the following we show that the reachability problem for MPDS restricted only to singly unbounded-budget computations is EXPTIME-complete.

Theorem 7: The $(\bar{k},\bar{d})$-budget-bounded reachability problem for MPDS is EXPTIME-complete if there is at most one index $i \in [1..n]$ such that either $k_i = \omega$ or $d_i = \omega$.

The rest of this section is devoted to the proof of Theorem 7.

Lower bound: It is known that the following problem is EXPTIME-complete [19]: Given a pushdown automaton $\mathcal{P}$ recognizing a language L, and $n-1$ finite state automata $\mathcal{A}_i$ recognizing languages L_i, check the non-emptiness of $L \cap \bigcap_{i=2}^{n} L_i$. We can show that this problem can be reduced, in polynomial time, to the reachability problem for MPDS restricted only to singly unbounded-budget computations. The idea is the following: The first stack (with unbounded number of contexts regardless of its depth) is used to simulate $\mathcal{P}$, while each other stack $i \in [2..n]$ is used to simulate the automaton $\mathcal{A}_i$. Each stack $i \in [1..n]$ has a depth bound 1 and a context budget 0. Moreover, the stack i contains at most one symbol which is the current state of $\mathcal{A}_i$. (We assume here that the automaton $\mathcal{A}_i$ does not contain ϵ-transitions.)

The simulation proceeds as follows: An ϵ-labeled transition of $\mathcal{P}$ is simulated by a transition of the first stack while the other stacks remain unchanged. A labeled transition of $\mathcal{P}$ with an input symbol a is simulated by a transition of the first stack, followed by a sequence of transitions in which the other stacks are checked and then updated, one after the other, to ensure that each $\mathcal{A}_i$ is able to perform a transition labeled by a.

Upper bound: To prove the upper bound, we reduce the reachability problem for $\mathcal{M}$ restricted to singly unbounded-budget computations to the non-emptiness test of a pushdown automaton whose size is exponential in $\mathcal{M}$. Recall that the non-emptiness test for pushdown automata is in PTIME [20]. In the following we use the same notations and definitions as in the previous subsection. We assume here that only the first stack can perform an unbounded number of contexts regardless of its depth. Then, we can show that M has a $(\bar{k},\bar{d})$-budget-bounded computation that reaches q_{final} iff there is a compatible sequence σ_1 of contexts of the first stack and for every stack $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$. In fact, we can prove that checking the existence of such a consistent word can be reduced, in exponential time, to the non-emptiness test of a pushdown automaton, and hence we obtain the completeness of Theorem 7.

Lemma 8: The problem of checking whether there is a compatible sequence σ_1 of contexts of the first stack and for every $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for the stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ can be reduced to the non-emptiness test of a pushdown automaton $\mathcal{P}$ whose size is exponential in $\mathcal{M}$.

Proof (sketch). We can construct, in polynomial time, a pushdown automaton $\mathcal{P}_1$ whose language $L(\mathcal{P}_1)$ is precisely the set of words $Summary(\sigma_1)$ where σ_1 is a compatible sequence of contexts of the first stack. On the other hand, as in the previous subsection, we can easily construct, in polynomial time, for every $i \in [2..n]$, a pushdown automaton $\mathcal{P}_i$ whose $(-1, d_i + k_i)$-bounded language is precisely $L_i(\mathcal{M})$. Then, checking whether there is a compatible sequence σ_1 of contexts of the first stack and for every $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for the stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\}, \ldots, \{Summary(\sigma_n)\})$ boils down to checking the non-emptiness of the language $L_0(\mathcal{P}_0) \cap \sqcup\!\sqcup(L(\mathcal{P}_1), L_{k_2+d_2}(\mathcal{P}_2), \ldots, L_{k_n+d_n}(\mathcal{P}_n))$. Finally, we can use standard automata constructions, to show that we can construct a pushdown automaton $\mathcal{P}$ such that $L(P) = L_0(\mathcal{P}_0) \cap \sqcup\!\sqcup(L(\mathcal{P}_1), L_{k_2+d_2}(\mathcal{P}_2), \ldots, L_{k_n+d_n}(\mathcal{P}_n))$. Moreover, the size of $\mathcal{P}$ is exponential in $\mathcal{M}$.

V. Other Subclasses of MPDS with Budget-Bounded Computations

In this section, we will briefly mention two other interesting subclasses of MPDS.

Definition 4 (Bounded stack-depth computations): We say that a $(\bar{k}, \bar{d})$-budget-bounded computation π is a *$\bar{d}$-bounded stack-depth* computation if and only if for every $i \in [1..n]$, we have $k_i = 0$ and $d_i \in \mathbb{N}$.

In the case of a bounded stack-depth computation, the size of the i-th stack in each reachable configuration in π is always bounded by d_i.

Definition 5 (Unbounded-budget computations): We say that a $(\bar{k}, \bar{d})$-budget-bounded computation π is an *unbounded-budget* computation if and only if there are at least two different stacks $i, j \in [1..n]$ such that $i \neq j$ and for every $\ell \in \{i, j\}$, either $k_\ell = \omega$ or $d_\ell = \omega$.

Observe that in the case of unbounded-budget computations, we have at least two different stacks that are allowed to perform an unbounded number of contexts regardless of their stack depth.

A. Known Results

In the following we recall some well-known results for the reachability problem for MPDS under budgets. More precisely, we consider bounded stack-depth and unbounded-budget computations.

Recall that, in the case of unbounded-budget computations, we have at least two different stacks that can perform unbounded number of context-switches regardless of their stack sizes. This implies that the reachability problem for MPDS restricted only to unbounded-budget computations is undecidable. This result can be shown using a reduction from the problem of checking non-emptiness of the intersection of two context-free languages (which is an undecidable problem).

Theorem 9: The $(\bar{k}, \bar{d})$-budget-bounded reachability problem for MPDS is undecidable if there are at least two different stacks $i, j \in [1..n]$ such that $i \neq j$ and for every $\ell \in \{i, j\}$, either $k_\ell = \omega$ or $d_\ell = \omega$.

One way to overcome this undecidability barrier is to bound the depth of each stack (which corresponds to case of MPDS restricted to bounded-stack-depth computations). In this case, we show:

Theorem 10: The $(\bar{k}, \bar{d})$-budget-bounded reachability problem for MPDS is PSPACE-complete if for every $i \in [1..n]$, we have $k_i = 0$ and $d_i \in \mathbb{N}$.

Proof: (*sketch*) Since in the case of a bounded stack-depth computation π, the depth of the i-th stack is bounded by d_i for any reachable configuration in π, the upper-bound of Theorem 10 can be obtained by an easy reduction to the emptiness problem for a Turing machine having n-tapes, and where each tape $i \in [1..n]$ has d_i cells.

The lower bound of Theorem 10 follows by a reduction from the non-emptiness test of the intersection of several regular languages (which is known to be PSPACE-hard). ■

VI. From Concurrent to Sequential

In this section, we will describe an automatic code-to-code translation from concurrent to sequential programs. The resulting sequential program simulates the concurrent program running under the uniformly bounded-budget restriction. First, we will briefly explain the language for concurrent programs. The remainder of the section describes the translation.

We consider a C-like programming language where concurrent programs consist of processes, procedures and statements. We assume that variables range over some (potentially infinite) data domain $\mathbb{D}$ and that we have a language of expressions $\langle expr \rangle$ interpreted over $\mathbb{D}$. The statements consists of simple C-like statements, enriched with `nop`, `assume`, `assert` and `atomic`. A *procedure* consists of a sequence of *arguments*, a set of *local variables*, and a sequence of *statements*. A *process* is a tuple $\mathcal{P} = \langle G, \mathcal{F}_1 \cdots \mathcal{F}_m \rangle$, where G is a finite set of *global variables* and each $\mathcal{F}_i$ is a procedure. For each process, there should be exactly one distinguished procedure called `main`, which constitutes the entry point of that process. A *concurrent program* is a tuple $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, consisting of a finite set S of *shared variables* and a sequence of processes.

Next, we describe an automatic transformation from a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$ to a sequential program $\mathcal{S}$ which simulates the behavior of $\mathcal{C}$ up to a given bound k_i of context switches for each $\mathcal{P}_i$ whenever the stack of $\mathcal{P}_i$ grows above d_i. If the stack of $\mathcal{P}_i$ *never* grows above d_i, there is no limit on the number of times $\mathcal{P}_i$ can be switched out.

A. Programs without Procedure Calls

Assume that we have a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, where no process $\mathcal{P}_i$ contains a procedure call, i.e. each process consists only of a main procedure. To construct the sequential program $\mathcal{S}$, we take each statement in the procedure and put it inside a scheduling loop. We introduce for each process $\mathcal{P}_i$ a variable `pc` $_i$ which keeps track of its programs counter. In the scheduling loop, each statement is enclosed in a conditional which contains a nondeterministic check of a Boolean variable `?` and a check for the correct program counter value. If the program counter check succeeds, but `?` happens to be false, the statement will not be executed. Additionally, all other program counter checks will fail, so the control flow will fall through the remainder of the statements. In this way, a context switch is simulated.

As an example, consider the program in Fig. 1 along with the sequential program which simulates it. It is easy to see that the sequential program simulates all behaviors of the concurrent program, including the interleaving `x = x + 2`, `x = 1`, `assert(x != 1)`, `x = 2` in which the assertion fails.

B. *Programs with Procedure Calls*

Assume now that we add procedure calls. There are two cases whenever a call happens in $\mathcal{P}_i$. Either the stack height is above d_i, in which case we must limit the number of preemptions of $\mathcal{P}_i$ to k_i as long as $\mathcal{P}_i$ stays above d_i, or the stack height is not above d_i, in which case the number of preemptions is unbounded. Instead of keeping track of these two possibilities, we will *inline* the procedure calls in the main procedure of each process $\mathcal{P}_i$ d_i times.

1) Inlining: For any process $\mathcal{P}$, let $I(\mathcal{P})$ be the result of inlining all procedure calls in the main procedure of $\mathcal{P}$. Note that this inlining might create new local variables. Let I^m denote the result of composing I with itself m times. Given a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, we construct an *inlined concurrent program* $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \cdots I^{d_n}(\mathcal{P}_n)\rangle$. In the execution of $\mathcal{C}'$, any procedure call in $I^{d_i}(\mathcal{P}_i)$ means that the corresponding execution in $\mathcal{C}$ would take the process $\mathcal{P}_i$ above its stack limit d_i. This means that we can differentiate between code based on whether it is inside or outside the main procedure of the process. Code that is outside the main procedure will be transformed in a way that takes into account the preemption bound k_i.

2) Context switching: In [13], La Torre, Madhusudan and Parlato describe a transformation that only keeps track of the local state of one process, at the expense of recomputing that state after context switches. More precisely, the transformation keeps track of $k + 1$ valuations $s_0 \cdots s_k$ of shared variables. The initial values of the shared variables are stored in s_0. Assume that process $\mathcal{P}_1$ starts running. When the context switch occurs, the values of the shared variables are stored in s_1. Another process then runs until there is another context switch, storing the shared varaibles in s_2. When $\mathcal{P}_1$ is switched in, it is executed *from the beginning* until the values of the shared variables equal s_1, i.e. the values when it was switched out. The shared variables are then assigned the values stored in s_2, and the execution continues. When the next context switch occurs, the shared variables are stored in s_3, and so on.

We use a similar approach to deal with context switches when a process $\mathcal{P}_i$ is above its stack bound d_i. The state of each process is thus stored explicitly up to the point where a process goes above its stack bound. When several processes are above their bounds, we only keep the local state of the one currently running. An important difference between our model and the one of [13] is that even when all processes are above their stack bound, we allow k preemptions *per process*. To facilitate this, we store $2k+1$ copies of the shared variables for each process.

3) Phases: An execution r of a single process in a concurrent program can be divided into a sequence $r_0, r_1, \ldots$ of executions separated by preemptions. We call each r_i a *phase*. In other words, a phase is a continuous sequence of statements. A process begins in r_0 and executes statements until there is a context switch. When the process gets switched back in, it runs r_1, and so on.

In the special case where a process is always above its stack bound, the execution of that process may consist of at most $k + 1$ phases. For this reason, we introduce for each process a variable `phase`, which keeps track of which phase the execution is in. This variable is increased whenever a context switch happens. Since we reconstruct the local state of a process by executing from the beginning, we also store a virtual phase `phase'`, which is updated both during the reconstruction and the actual execution. This means that as long as `phase'< phase`, we are reconstructing the local state.

In general, a process is not always above its stack bound. When a process goes below that bound, the budget of allowed preemptions is reset. In our transformation, this means that we reset the phase variables, starting again from `phase = 0` the next time a procedure call happens.

4) Transformation: For a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n\rangle$, we first construct the corresponding inlined concurrent program $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \cdots I^{d_n}(\mathcal{P}_n)\rangle$. We then transform $\mathcal{C}'$ into a sequential program $\mathcal{S}$ that simulates $\mathcal{C}'$. We can find among the global variables of $\mathcal{S}$, for each process $\mathcal{P}_t$, the sets $S_t^0, \ldots, S_t^{2k+1}$ of copies of the shared variables of $\mathcal{C}$. The transformation of the statements in the main procedures of each process is done in the same way as previously, with the exception of *procedure calls*. Before each procedure call in a process $\mathcal{P}_t$, we insert a code block that, if the process is not recomputing the local state, saves the current values of the global variables in S_t^0. This code block is shown in Fig. 2.

The set of procedures of the sequential program is the union of the transformed procedures of its processes. When we transform a procedure, we perform three steps:

- To simulate context switches, we add the code shown in the right side in Fig. 2 before any statement that contains shared variables and therefore is visible to the outside.
- Before any statement that contains shared variables, we also add code to detect whether the local context has been reconstructed or not. This code is shown in Fig. 3.
- At the end of the procedure, we check if we are about to return to the scheduling loop without having reconstructed the local state. In this case, we abort.

VII. Experimental Results

We have evaluated our approach on several examples, including one in which a big number of context switches is needed in order to reach a bad state. The experiments presented in Fig. 4 were run on a 2.2 GHz Intel Core i7 with 4 GB of memory. Most literature examples are written in pseudo code or C-like code. In order to run them, we manually translated them to our syntax. This operation can be automated. It is in fact possible to extend our tool in order to parse C code.

We have implemented our code-to-code translation scheme in Haskell [21]. The scheme inputs a concurrent program P and produces a sequential program P' such that running P under the uniformly bounded-budget restriction yields the same set of reachable states as running P'. The sequential program is delivered in different languages, namely: Remopla for Moped [8], and a C-like language for Cbmc [7] and Esbmc [9]. We use these three tools as back end to verify the obtained sequential code. Our experimental results are

```
process example:

int x = 0;

process p1:
void main(){
  x = 1;
  x = 2;
}

process p2:
void main(){
  x = x + 2;
  assert(x != 1);
}
```

```
process transformed:

int pc1 = 1;
int pc2 = 1;
int running;
int x = 0;

void scheduler(){
  while(progress){
    progress = false;

    // schedule a process
    if( ? && pc1!=3){
      running = 1;
    }
    if( ? && pc2!=3) {
      running = 2;
    }
    // process 1
    if(running == 1){
      if(pc1==1 && ?){
        x = 1;
        progress = true;
        pc1 = 2;
      }
      if(pc1==2 && ?){
        x = 2;
        progress = true;
        pc1 = 3;
      }
    }
    // process 2
    if(running == 2){
      if(pc2==1 && ?){
        x = x + 2;
        progress = true;
        pc2 = 2;
      }
      if(pc2==2 && ?){
        assert(x != 1);
        progress = true;
        pc2 = 3;
      }
    }
  }
}
```

Fig. 1. Left: Transformation of Procedure Calls in Process t. Right: Context Switches in Procedures of Process t.

```

if(phase_t == 0){
  S_t^0 = S;
  S = S_t^0;
}
  ⋮
if(phase_t == k){
  S_t^2k = S;
  S = S_t^0;
}
```

```

if(!ret && ?){
  if(phase'_t == phase_t){
    if(phase_t == 0){
      S_t^1 = S;
    }
    ⋮
    if(phase_t == k){
      S_t^{2k+1} = S;
    }

    phase_t =
    phase_t + 1;
    ret = true;
}
```

Fig. 2. Left: Transformation of Procedure Calls in Process t. Right: Context Switches in Procedures of Process t.

```
if(!ret && ?){
  if(phase'_t < phase_t){
    if(phase'_t == 0 && S == S_t^1){
      phase'_t = 1;
      S = S_t^2;
    }
    ⋮
    if(phase_t == k-1 && S == S_t^{2k-1}){
      phase'_t = k;
      S = S_t^2k;
    }
}
```

Fig. 3. Checking Reconstruction of Local State in Process t

Examples		Type of Analysis							
		Concurrent to Sequential				Concurrent			
		k_1	MOPED	CBMC	ESBMC	k_2	POIROT	k_3	ESBMC
Account	[22]	0	-.-	-.-	**1.1**	4	3.34	10	-.-
BigNum	[21]	0	**8.39**	-.-	-.-	26	-.-	234	-.-
Bluetooth3a	[21]	0	744.49	-.-	-.-	11	**18.38**	28	*FP*
Token Ring	[22]	0	**0.13**	0.2	0.18	1	2.72	4	1.47
Account Bad	[22]	0	-.-	0.48	1.41	1	2.13	4	**0.11**
BigNum Bad	[21]	0	**5.9**	13.4	239.4	26	-.-	26	-.-
Bluetooth1	[23]	1	4.18	**0.37**	1.28	2	1.92	5	*NF*
Bluetooth2	[23]	1	0.64	5.68	34.38	3	2.9	5	**0.5**
Bluetooth3b	[23]	1	1.26	**0.95**	5.0	2	2.5	5	*NF*
Infinite Loop 1	[24]	1	4.1	0.2	0.25	2	1.45	1	**0.08**
Infinite Loop 2	[24]	1	17.3	0.84	3.85	1	0.96	1	**0.09**
Token Ring Bad	[22]	0	**0.13**	0.16	0.26	2	2.74	4	0.27

Fig. 4. We report the running times of our experimentation results in seconds. We use the symbol -.- to denote a *timeout* (set to 900 seconds). The column k_1 contains the context-switch budget for our code-to-code translation. The columns k_2 and k_3 are the number of context switches given as input for POIROT and ESBMC respectively. NF: Bug Not Found. FP: False Positive.

then compared to ones obtained using two verification tools for concurrent programs, namely ESBMC and POIROT [10]. The time required for sequentialization is negligible and not included in the results.

The table in Fig. 4 summarizes our experimental results. In the upper part of the table, only safe (correct) programs are considered. We fixed the context switch bounds k_1, k_2, and k_3 such that all compared tools are able to cover the same set of control locations. The results show that our approach manages to perform better in three out of four examples, in particular for the BIGNUM example where the concurrent tools timeout. In this example, a large number of context switches (26) is required to find the assertion violation. Also, we noticed that ESBMC finds a bug in the correct example BLUETOOTH3A. It has been confirmed that this is a false positive [25]. In the lower part of the table, we consider faulty programs. For half of those programs, the experimental results show that our approach succeeds in finding all the bugs within a smaller amount of time compared to the concurrent tools. In particular, both POIROT and ESBMC timed out on the BIGNUM BAD example. Also, ESBMC, which did as well as our approach in terms of time, failed to find bugs in the faulty examples BLUETOOTH 1 and 3*b* regardless of the number of context-switches it was allowed.

VIII. CONCLUSION

We have introduced the class of MPDS with budgets where each stack can perform an unbounded number of context switches if its size is below or equal to a given bound, while it is restricted to a finite number of context switches when its size is above that bound. We have identified two decidable subclasses of MPDS with budgets, namely uniformly bounded-budget MPDS and singly unbounded-budget MPDS. We have shown that the reachability problem for uniformly bounded-budget MPDS and singly unbounded-budget MPDS is respectively PSPACE-complete and EXPTIME-complete. Moreover, we have proposed a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that, running P under the uniformly bounded-budget restriction yields the same set of reachable states as running P'. We have implemented a prototype tool, and run it successfully on a set of benchmarks.

References

[1] S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 93–107.

[2] S. La Torre, P. Madhusudan, and G. Parlato, “A robust class of context-sensitive languages,” in *LICS*. IEEE, 2007, pp. 161–170.

[3] M. F. Atig, B. Bollig, and P. Habermehl, “Emptiness of multi-pushdown automata is 2ETIME-complete,” in *DLT’08*, ser. LNCS, vol. 5257. Springer, 2008, pp. 121–133.

[4] M. Lange and H. Leiß, “To CNF or not to CNF ? An efficient yet presentable version of the CYK algorithm,” *Informatica Didactica*, vol. 8, 2008-2010.

[5] S. La Torre and M. Napoli, “Reachability of multistack pushdown systems with scope-bounded matching relations,” in *CONCUR*, ser. LNCS, vol. 6901. Springer, 2011, pp. 203–218.

[6] M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs,” in *PLDI*. ACM, 2007, pp. 446–455.

[7] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *TACAS*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[8] J. Esparza, S. Kiefer, and S. Schwoon, “Abstraction refinement with Craig interpolation and symbolic pushdown systems,” in *TACAS*, ser. LNCS, vol. 3920, 2006, pp. 489–503.

[9] L. Cordeiro, J. Morse, D. Nicole, and B. F. 0002, “Context-bounded model checking with esbmc 1.17 - (competition contribution).” in *TACAS*, ser. LNCS, vol. 7214, 2012, pp. 534–537.

[10] S. Lahiri, A. Lal, and S. Qadeer, “Poirot,” microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/projects/poirot

[11] A. Finkel and A. Sangnier, “Reversal-bounded counter machines revisited,” in *MFCS*, ser. LNCS, vol. 5162. Springer, 2008, pp. 323–334.

[12] A. Lal and T. W. Reps, “Reducing concurrent analysis under a context bound to sequential analysis,” *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.

[13] S. La Torre, P. Madhusudan, and G. Parlato, “Reducing context-bounded concurrent reachability to sequential reachability,” in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 477–492.

[14] ——, “Model-checking parameterized concurrent programs using linear interfaces,” in *CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 629–644.

[15] M. Emmi, S. Qadeer, and Z. Rakamarić, “Delay-bounded scheduling,” in *POPL*. ACM, 2011, pp. 411–422.

[16] S. La Torre and G. Parlato, “Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width,” University of Southampton, Technical Report, march 2012.

[17] G. Parlato, personal communication.

[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[19] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre, “Reachability analysis of communicating pushdown systems,” in *FOSSACS*, ser. LNCS, vol. 6014. Springer, 2010, pp. 267–281.

[20] A. Bouajjani, J. Esparza, and O. Maler, “Reachability analysis of pushdown automata: Application to model-checking,” in *CONCUR*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[21] May 2012. [Online]. Available: http://user.it.uu.se/%7Ejarst116/fmcad2012/

[22] “Esbmc concurrency benchmark,” Feb. 2009. [Online]. Available: http://users.ecs.soton.ac.uk/lcc08r/esbmc/concurrent-software-benchmarks.zip

[23] D. Suwimonteerabuth, “Reachability in pushdown systems: Algorithms and applications,” Ph.D. dissertation, Technische Universität München, 2009.

[24] S. Qadeer, S. K. Rajamani, and J. Rehof, “Summarizing procedures in concurrent programs,” 2004.

[25] J. Morse, personal communication.

Quantifier Elimination by Dependency Sequents

Eugene Goldberg and Panagiotis Manolios
Northeastern University, USA, {eigold,pete}@ccs.neu.edu

***Abstract*—We consider the problem of existential quantifier elimination for Boolean CNF formulas. We present a new method for solving this problem called Derivation of Dependency-Sequents (DDS). A Dependency-sequent (D-sequent) is used to record that a set of quantified variables is redundant under a partial assignment. We introduce the *join* operation that produces new D-sequents from existing ones. We show that DDS is compositional, *i.e.*, if our input formula is a conjunction of independent formulas, DDS automatically recognizes and exploits this information. We introduce an algorithm based on DDS and present experimental results demonstrating its potential.**

I. Introduction

In this paper, we consider the problem of eliminating quantifiers from formulas of the form $\exists X[F]$ where F is a Boolean CNF formula and some variables of F may be free of quantifiers. 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 $G \equiv \exists X[F]$.

Our interest in the QE problem is twofold. First, the QE problem occurs in numerous areas of hardware design and verification, *e.g.*, in symbolic model checking [9], [10], [22] when computing reachable states. Second, one can argue that progress in solving the QE problem should have a deep impact on SAT-solving [15]. In particular, as McMillan pointed out, even the basic operation of resolution is related to the QE problem [23]. The resolvent C of clauses C',C'' on a variable v is obtained by eliminating the quantifier from $\exists v[C' \wedge C'']$.

The success of resolution-based SAT-solvers [21], [24] has led to the hunt for efficient SAT-based algorithms for the QE problem [23], [18], [6], [13]. In this paper, we continue in this direction by introducing a resolution-based QE algorithm operating on CNF formulas. Such formulas are ubiquitous in hardware verification because a circuit N can be represented by a CNF formula whose size is linear in that of N and that has the same set of variables as N.

Our approach is based on the following observation. The QE problem is trivial if F does not depend on variables of X. In this case, dropping the quantifiers from $\exists X[F]$ produces an equivalent formula. If F depends on X, after adding to F a set of clauses implied by F, the variables of X may become redundant in $\exists X[F]$. That is, the clauses of F depending on X can be dropped and the resulting CNF formula G is equivalent to the original formula $\exists X[F]$. The problem is that one needs to know *when the variables of X become redundant*.

Unfortunately, resolution is deficient in expressing redundancy of variables. Let $\boldsymbol{y}$ be an assignment to all non-quantified variables of $\exists X[F]$. Let $F_{\boldsymbol{y}}$ denote F under assignment $\boldsymbol{y}$. If $F_{\boldsymbol{y}}$ is unsatisfiable, then a clause C falsified by $\boldsymbol{y}$ can be derived by resolving clauses of F. After adding C to F, the variables of X are redundant in $\exists X[F_{\boldsymbol{y}}]$. In this case, resolution works. Assume, however, that $F_{\boldsymbol{y}}$ is *satisfiable*. Then, the variables of X are *also redundant* in $\exists X[F_{\boldsymbol{y}}]$ because $F_{\boldsymbol{y}}$ remains satisfiable after removing any set of clauses. But a resolution derivation cannot express this fact since no clause falsified by $\boldsymbol{y}$ is implied by F.

To address this problem, we introduce the notion of Dependency sequents (*D-sequents*). A D-sequent has 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 states that in the subspace specified by $\boldsymbol{q}$, the variables of Z are redundant in $\exists X[F]$. That is, in this subspace, after dropping clauses with variables of Z from F one gets a formula equivalent to $\exists X[F]$. In particular, the D-sequent $(\exists X[F], \boldsymbol{y}) \rightarrow X$ holds, if formula $F_{\boldsymbol{y}}$ is satisfiable where $\boldsymbol{y}$ is an assignment to the non-quantified variables of $\exists X[F]$.

In this paper, we introduce a QE algorithm called *DDS* (Derivation of D-Sequents). In *DDS*, adding resolvent clauses to F is accompanied by computing D-sequents. The algorithm terminates when the D-sequent $(\exists X[G], \emptyset) \rightarrow X$ is derived, where G is a CNF formula that includes the initial clauses, F, and some resolvent clauses. Upon termination, the variables of X are unconditionally redundant and a solution to the QE problem is obtained by dropping the clauses containing variables of X from G.

DDS includes a *join* operation that generates new D-sequents from existing ones. Let $(\exists X[F], \boldsymbol{q'}) \rightarrow Z$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow Z$ be valid D-sequents where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ have opposite assignments to exactly one variable v. Then a new, valid D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ can be obtained by joining the D-sequents above, where $\boldsymbol{q}$ contains all assignments of $\boldsymbol{q'}$ and $\boldsymbol{q''}$ except those to v.

In this paper, we compare *DDS* with its counterparts both theoretically and experimentally. In particular, we show that *DDS* is *compositional* while algorithms based on enumeration of satisfying assignments [23], [19], [13], [6] are not. Compositionality here means that given an $\exists$CNF formula $\exists X[F_1 \wedge \cdots \wedge F_k]$ where formulas F_i depend on non-overlapping sets of variables, *DDS* breaks the QE problem into k independent subproblems. *DDS* is a branching algorithm and yet it remains compositional no matter how branching variables are chosen. Compositionality of *DDS* means that its performance can be *exponentially better* than that of enumeration-based QE algorithms. Since *DDS* is a branching algorithm it can process variables of different branches in different orders. This gives *DDS* a big edge over QE algorithms that eliminate quantified variables one by one using a global order [18], [15].

D-sequents are related to boundary points [14]. A boundary point is a complete assignment to variables of F with certain properties. To make variables of $Z \subseteq X$ redundant in $\exists X[F]$ one needs to eliminate a particular set of boundary points. This elimination is performed by adding to F resolvent clauses that do not depend on variables of Z. Although, *DDS* does not compute boundary points *explicitly*, we introduce them in this paper for the following two reasons. First, boundary points provide the semantics of *DDS*. In particular, the proofs of many propositions we use in this paper are based on the notion of boundary points. Second, *DDS* avoids an explicit computation of boundary points by using a particular branching order: non-quantified variables of $\exists X[F]$ are assigned before quantified. However, there is no guarantee that such an order is always optimal and so to achieve the best performance one may need to interleave assignments to quantified and non-quantified variables. In this case, to reduce the number of new resolvent clauses to be added to F, one, in general, cannot avoid an explicit computation of boundary points [17].

The contribution of this paper is as follows. First, we relate the notion of variable redundancy with the elimination of boundary points. Second, we introduce the notion of D-sequents and the operation of joining D-sequents. Third, we describe *DDS*, a QE algorithm; we prove its correctness and evaluate it experimentally. Fourth, we show that *DDS* is compositional.

This paper is structured as follows. In Section II, we relate the notions of variable redundancy and boundary points. Section III explains the strategy of *DDS* in terms of boundary point elimination. D-sequents are introduced in Section IV. Sections V and VI describe *DDS* and discuss its compositionality. Section VII gives experimental results. Background is discussed in Section VIII, and conclusions are presented in Section IX.

II. Redundant Variables, Boundary Points and Quantifier Elimination

The main objective of this section is to introduce the notion of redundant variables and to relate it to the elimination of removable boundary points.

A. Redundant Variables and Quantifier Elimination

Definition 1: An $\exists$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. $\mathit{Vars}(\boldsymbol{q})$ denotes the variables assigned in $\boldsymbol{q}$; $\mathit{Vars}(F)$ denotes the set of variables of F; $\mathit{Vars}(C)$ denotes the variables of C; and $\mathit{Vars}(\exists X[F]) = \mathit{Vars}(F) \setminus X$.

Definition 2: Let C be a clause, H be a formula, and $\boldsymbol{q}$ be an assignment. $C_{\boldsymbol{q}}$ is *true* if C is satisfied by $\boldsymbol{q}$; otherwise it is the clause obtained from C by removing all literals falsified by $\boldsymbol{q}$. Let $\boldsymbol{p}$ be $\boldsymbol{q} \cap \mathit{Vars}(H)$. $H_{\boldsymbol{q}}$ denotes the formula obtained from H by first removing the clauses of H satisfied by $\boldsymbol{p}$, and then removing all the literals falsified by $\boldsymbol{p}$ in the remaining clauses of H. If $\mathit{Vars}(H) \subseteq \mathit{Vars}(\boldsymbol{q})$, then $H_{\boldsymbol{q}}$ is semantically equivalent to a constant, and in the sequel, we will make use of this without explicit mention.

Definition 3: Let G, H be formulas. We say that G, H are *equivalent*, written $G \equiv H$, if for all assignments, $\boldsymbol{q}$, such that $\mathit{Vars}(\boldsymbol{q}) \supseteq (\mathit{Vars}(G) \cup \mathit{Vars}(H))$, we have $G_{\boldsymbol{q}} = H_{\boldsymbol{q}}$. Notice that $G_{\boldsymbol{q}}$ and $H_{\boldsymbol{q}}$ have no free variables, so by $G_{\boldsymbol{q}} = H_{\boldsymbol{q}}$ we mean semantic equivalence.

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: A clause C of F is called a $\boldsymbol{Z}$**-clause** if $\mathit{Vars}(C) \cap Z \neq \emptyset$. Denote by $\boldsymbol{F^Z}$ the set of all Z-clauses of F.

Definition 6: The variables of Z are **redundant** in CNF formula F if $F \equiv (F \setminus F^Z)$. The variables of Z are **redundant** in $\exists$CNF formula $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus F^Z]$. We note that since $F \setminus F^Z$ does not contain any Z variables, we could have written $\exists (X \setminus Z)[F \setminus F^Z]$. To simplify notation, we avoid explicitly using this optimization in the rest of the paper.

B. Redundant Variables and Boundary Points

Definition 7: Given assignment $\boldsymbol{p}$ and a formula F, we say that $\boldsymbol{p}$ is an F-**point** (or a **point** of F) if $\mathit{Vars}(F) \subseteq \mathit{Vars}(\boldsymbol{p})$.

In the sequel, by "assignment" we mean a possibly partial one. To refer to a *complete* assignment we will use the term "point".

Definition 8: A point $\boldsymbol{p}$ of CNF formula F is called a $\boldsymbol{Z}$**-boundary point** of F if a) $Z \neq \emptyset$ and b) $F_{\boldsymbol{p}} = \mathit{false}$ and c) every clause of F falsified by $\boldsymbol{p}$ is a Z-clause and d) the previous condition breaks for every proper subset of Z.

The term "boundary" is justified as follows. Let F be a satisfiable CNF formula with at least one clause. Then there always exists a $\{x\}$-boundary point of F, $x \in \mathit{Vars}(F)$ that is different from a satisfying assignment only in value of x.

Definition 9: Given a CNF formula F and a Z-boundary point, $\boldsymbol{p}$, of F:

- $\boldsymbol{p}$ is X-**removable** in F if 1) $Z \subseteq X \subseteq \mathit{Vars}(F)$; and 2) there is a clause C such that a) $F \Rightarrow C$; b) $C_{\boldsymbol{p}} = \mathit{false}$; and c) $\mathit{Vars}(C) \cap X = \emptyset$.
- $\boldsymbol{p}$ is **removable** in $\exists X[F]$ if $\boldsymbol{p}$ is X-removable in F.

In the above definition, notice that $\boldsymbol{p}$ is not a Z-boundary point of $F \wedge C$ because $\boldsymbol{p}$ falsifies C and $\mathit{Vars}(C) \cap Z = \emptyset$.

Proposition 1: A Z-boundary point $\boldsymbol{p}$ of F is removable in $\exists X[F]$, iff one cannot turn $\boldsymbol{p}$ into an assignment satisfying F by changing only the values of variables of X.

The proofs are given in [16].

Proposition 2: The variables of $Z \subseteq X$ are not redundant in $\exists X[F]$ iff there is an X-removable W-boundary point of F, $W \subseteq Z$.

Proposition 2 justifies the following strategy of solving the QE problem. Add to F a set G of clauses that a) are implied by F; b) eliminate all X-removable Z-boundary points for all $Z \subseteq X$. By dropping all X-clauses of F, one produces a solution to the QE problem.

III. Boundary Points And Divide-And-Conquer Strategy

In this section, we provide the semantics of the QE algorithm *DDS* described in Section V. *DDS* is a branching algorithm. Given an $\exists$CNF formula $\exists X[F]$, it branches on variables of F until proving redundancy of variables of X in the current subspace becomes trivial. Then *DDS* merges the results obtained in different branches to prove that the variables of X are redundant in the entire search space.

Below we give propositions justifying the divide-and-conquer strategy of *DDS*. Proposition 3 shows how to perform elimination of removable boundary points of F in the subspace specified by assignment $\boldsymbol{q}$. Proposition 4 justifies proving redundancy of variables of X in subspace $\boldsymbol{q}$ one by one. Finally, Subsection III-B describes two cases where proving variable redundancy is trivial.

A. Decomposing the Problem of Boundary Point Elimination

Definition 10: Let $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$ be assignments. The expression $\boldsymbol{q_1} \leq \boldsymbol{q_2}$ denotes the fact that $\mathit{Vars}(\boldsymbol{q_1}) \subseteq \mathit{Vars}(\boldsymbol{q_2})$ and each variable of $\mathit{Vars}(\boldsymbol{q_1})$ has the same value in $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$.

Proposition 3: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$. Let $\boldsymbol{p}$ be a Z-boundary point of F where $\boldsymbol{q} \leq \boldsymbol{p}$ and $Z \subseteq X$. Then if $\boldsymbol{p}$ is removable in $\exists X[F]$ it is also removable in $\exists X[F_{\boldsymbol{q}}]$.

The opposite is not true: a boundary point may be X-removable in $F_{\boldsymbol{q}}$ and not X-removable in F. For instance, if $X = \mathit{Vars}(F)$, a Z-boundary point $\boldsymbol{p}$ of F is removable in $\exists X[F]$ for any $Z \subseteq X$ only by adding an empty clause to F. So if F is satisfiable, $\boldsymbol{p}$ is not removable. Yet $\boldsymbol{p}$ may be removable in $\exists X[F_{\boldsymbol{q}}]$ if $F_{\boldsymbol{q}}$ is unsatisfiable.

Definition 11: Let $\exists X[F]$ be an $\exists$CNF formula, $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$, and $Z \subseteq (X \setminus \mathit{Vars}(\boldsymbol{q}))$. Variables of Z are called **virtually redundant** in $\exists X[F_{\boldsymbol{q}}]$ if $\exists X[F_{\boldsymbol{q}} \setminus (F_{\boldsymbol{q}})^Z] \equiv (\exists X[F])_{\boldsymbol{r}}$ where $\boldsymbol{r} \leq \boldsymbol{q}$ and $\mathit{Vars}(\boldsymbol{r}) = \mathit{Vars}(\boldsymbol{q}) \setminus X$.

Redundancy of variables of Z in $\exists X[F_{\boldsymbol{q}}]$ in terms of Definition 6 is a special case of virtual redundancy. To prove variables of Z redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$, it is sufficient to show virtual redundancy of Z in $\exists X[F_{\boldsymbol{q}}]$. The reason is that one can ignore Z-boundary points that are removable in $\exists X[F_{\boldsymbol{q}}]$ and not removable in $\exists X[F]$. We introduce a new notion of redundancy of variables Z in $F_{\boldsymbol{q}}$ because the operation of joining D-sequents (Definition 16) preserves only virtual redundancy of Z. In the sequel, when we say that variables of Z are redundant in $\exists X[F_{\boldsymbol{q}}]$ we mean that they are **at least virtually redundant**.

Proposition 4: Let $\exists X[F]$ be a CNF formula and $\boldsymbol{q}$ be an assignment to variables of F. Let the variables of Z be redundant in $\exists X[F_{\boldsymbol{q}}]$ where $Z \subseteq (X \setminus \mathit{Vars}(\boldsymbol{q}))$. Let a variable x of $X \setminus (\mathit{Vars}(\boldsymbol{q}) \cup Z)$ be redundant in $\exists X[F_{\boldsymbol{q}} \setminus (F_{\boldsymbol{q}})^Z]$. Then the variables of $Z \cup \{x\}$ are redundant in $\exists X[F_{\boldsymbol{q}}]$.

Proposition 4 shows that one can make variables of $X \setminus \mathit{Vars}(\boldsymbol{q})$ redundant *incrementally*, if every $\{x\}$-clause is removed from $F_{\boldsymbol{q}}$ as soon as variable x is proved redundant.

B. Two Trivial Cases of Variable Redundancy

Definition 12: Let C' and C'' be clauses having opposite literals of exactly one variable $v \in \mathit{Vars}(C') \cap \mathit{Vars}(C'')$. 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 13: A variable x of a CNF formula F is called **blocked** if no two clauses of F are resolvable on x. A variable x is called **monotone** if it is a pure literal variable [11] (i.e. literals of only one polarity of x are present in F). A monotone variable is a special case of a blocked variable.

The notion of blocked variables is related to that of blocked clauses introduced in [20] (not to confuse with *blocking* clauses [23]). A clause C of F is blocked with respect to x if no clause C' of F is resolvable with C on x. Variable x is blocked in F if every $\{x\}$-clause of F is blocked with respect to x.

Proposition 5: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$. Let a variable x of $X \setminus \mathit{Vars}(\boldsymbol{q})$ be blocked in $F_{\boldsymbol{q}}$. Then x is redundant in $\exists X[F_{\boldsymbol{q}}]$.

Proposition 6: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$. Let $F_{\boldsymbol{q}}$ have an empty clause. Then the variables of $X \setminus \mathit{Vars}(\boldsymbol{q})$ are redundant in $\exists X[F_{\boldsymbol{q}}]$.

IV. Dependency Sequents (D-sequents)

In this section, we define D-sequents and introduce the operation of joining D-sequents.

A. Definition of D-sequents

Definition 14: Let $\exists X[F]$ be an $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$ and Z be a subset of $X \setminus \mathit{Vars}(\boldsymbol{q})$. A dependency sequent (**D-sequent**) has the form $(\exists X[F], \boldsymbol{q}) \rightarrow Z$. It states that the variables of Z are redundant in $\exists X[F_{\boldsymbol{q}}]$.

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 $F_{\boldsymbol{q}}$ = C_2 because C_1 is satisfied. Notice that x is monotone and so redundant in $F_{\boldsymbol{q}}$ (Proposition 5). Hence, the D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow \{x\}$ holds.

According to Definition 14, a D-sequent holds with respect to a particular $\exists$CNF formula $\exists X[F]$. Proposition 7 shows that this D-sequent also holds after adding to F resolvent clauses.

Proposition 7: Let $\exists X[F]$ be an $\exists$CNF formula. Let $H = F \wedge G$ where $F \Rightarrow G$. Let $\boldsymbol{q}$ be an assignment to $\mathit{Vars}(F)$. Then if $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ holds, $(\exists X[H], \boldsymbol{q}) \rightarrow Z$ does too.

B. Join Operation for D-sequents

In this subsection, we introduce the operation of joining D-sequents. The join operation produces a new D-sequent from two D-sequents derived earlier. The semantics of this operation in terms of elimination of boundary points is quite simple. Let A_1 and A_2 be subspaces from which all removable boundary points of F relevant to redundancy of $Z \subseteq X$ in $\exists X[F]$ have been eliminated. The join operation produces a new subspace A such that $A \subseteq A_1 \cup A_2$. We start with introducing resolution of assignments that is similar to that of clauses.

```
// $\xi$ denotes $\exists X[F]$, $\boldsymbol{q}$ is an assignment to $Vars(F)$
// $\Omega$ denotes a set of active D-sequents

$DDS(\xi,\boldsymbol{q},\Omega)$ {
1  $(\Omega, ans, C) \leftarrow atomic\_D\_seqs(\xi,\boldsymbol{q},\Omega)$;
2  if ($ans$ = $sat$) return($\xi,\Omega, sat$);
3  if ($ans$ = $unsat$) return($\xi,\Omega, unsat, C$);
4  $v := pick\_variable(F,\boldsymbol{q},\Omega)$;
5  $(\xi,\Omega, ans_0, C_0) \leftarrow DDS(\xi,\boldsymbol{q} \cup \{(v=0)\},\Omega)$;
6  $(\Omega^{sym},\Omega^{asym}) \leftarrow split(F,\Omega,v)$;
7  if ($\Omega^{asym} = \emptyset$) return($\xi,\Omega, ans_0, C_0$);
8  $\Omega := \Omega \setminus \Omega^{asym}$;
9  $(\xi,\Omega, ans_1, C_1) \leftarrow DDS(\xi,\boldsymbol{q} \cup \{(v=1)\},\Omega)$;
10 if (($ans_0 = unsat$) and ($ans_1 = unsat$)){
11     $C := resolve\_clauses(C_0, C_1, v)$;
12     $F := F \wedge C$;
13     $\Omega := process\_unsat\_clause(\xi, C, \Omega)$;
14     return($\xi,\Omega, unsat, C$);}
15 $\Omega := merge(\xi,\boldsymbol{q}, v, \Omega^{asym}, \Omega)$;
16 return($\xi,\Omega, sat$);}
```

Fig. 1. *DDS* procedure

Definition 15: 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 8: Let $\exists X[F]$ be an $\exists$CNF formula. Let D-sequents $(\exists X[F],\boldsymbol{q'}) \rightarrow Z$ and $(\exists X[F],\boldsymbol{q''}) \rightarrow Z$ 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, the D-sequent $(\exists X[F],\boldsymbol{q}) \rightarrow Z$ holds too.

Definition 16: We will say that the D-sequent $(\exists X[F],\boldsymbol{q}) \rightarrow Z$ of Proposition 8 is produced by **joining D-sequents** $(\exists X[F],\boldsymbol{q'}) \rightarrow Z$ and $(\exists X[F],\boldsymbol{q''}) \rightarrow Z$ at v.

V. Description of *DDS*

In this section, we describe a QE algorithm called *DDS* (Derivation of D-Sequents). *DDS* derives D-sequents $(\exists X[F],\boldsymbol{q}) \rightarrow \{x\}$ stating the redundancy of one variable of X. From now on, we will use a short notation of D-sequents writing $\boldsymbol{q} \rightarrow \{x\}$ instead of $(\exists X[F],\boldsymbol{q}) \rightarrow \{x\}$. We will assume that the parameter $\exists X[F]$ missing in $\boldsymbol{q} \rightarrow \{x\}$ is the *current* $\exists$CNF formula (with all resolvent clauses added to F so far). One can omit $\exists X[F]$ from D-sequents because from Proposition 7 it follows that once D-sequent $(\exists X[F],\boldsymbol{q}) \rightarrow \{x\}$ is derived it holds after adding to F any set of resolvent clauses. We will call D-sequent $\boldsymbol{r} \rightarrow \{x\}$ **active** in the branch specified by assignment $\boldsymbol{q}$ if $\boldsymbol{r} \leq \boldsymbol{q}$ i.e. if this D-sequent provides a proof of redundancy of x in subspace $\boldsymbol{q}$.

A description of *DDS* is given in Figure 1. *DDS* accepts an $\exists$CNF formula $\exists X[F]$ (denoted as ξ), an assignment $\boldsymbol{q}$ to $Vars(F)$ and a set Ω of active D-sequents stating redundancy of *some* variables of $X \setminus Vars(\boldsymbol{q})$ in $\exists X[F_{\boldsymbol{q}}]$. *DDS* returns a modified formula $\exists X[F]$ (where resolvent clauses have been added to F) and a set Ω of active D-sequents stating redundancy of *every* variable of $X \setminus Vars(\boldsymbol{q})$ in $\exists X[F_{\boldsymbol{q}}]$. *DDS* also returns the answer *sat* if $F_{\boldsymbol{q}}$ is satisfiable. If $F_{\boldsymbol{q}}$ is unsatisfiable, *DDS* returns the answer *unsat* and a clause of F falsified by $\boldsymbol{q}$. To build a CNF formula equivalent to ξ,

```
$atomic\_D\_seqs(\xi,\boldsymbol{q},\Omega)$ {
1  if ($\exists$ clause $C \in F$ falsif. by $\boldsymbol{q}$){
2     $\Omega := process\_unsat\_clause(\xi, C, \Omega)$;
3     return($\Omega$, $unsat, C$);}
4  $\Omega$:=$new\_redund\_vars(\xi,\boldsymbol{q},\Omega)$;
5  if ($all\_unassgn\_vars\_redund(\xi,\boldsymbol{q},\Omega)$) return($\Omega$, $sat$);
6  return($\Omega$, $unknown$)};
```

Fig. 2. *atomic_D_seqs* procedure

one needs to call *DDS* with $\boldsymbol{q} = \emptyset$, $\Omega = \emptyset$ and discard the X-clauses of the CNF formula F returned by *DDS*.

A. The Big Picture

First, *DDS* looks for variables whose redundancy is trivial to prove (lines 1-3). If some variables of $X \setminus Vars(\boldsymbol{q})$ are not proved redundant yet, *DDS* picks a branching variable v (line 4). Then it extends $\boldsymbol{q}$ by assignment $(v = 0)$ and recursively calls itself (line 5) starting the left branch of v. Once the left branch is finished, *DDS* extends $\boldsymbol{q}$ by $(v = 1)$ and explores the right branch (line 9). The results of the left and right branches are then merged (lines 10-16).

DDS terminates when for every variable x of $X \setminus Vars(\boldsymbol{q})$ it derives a D-sequent $\boldsymbol{g} \rightarrow \{x\}$ where $\boldsymbol{g} \leq \boldsymbol{q}$. According to Proposition 4, derivation of such D-sequents means that the D-sequent $\boldsymbol{q} \rightarrow X \setminus Vars(\boldsymbol{q})$ holds. Proposition 4 is applicable here because once a variable x of $X \setminus Vars(\boldsymbol{q})$ is proved redundant in $\exists X[F_{\boldsymbol{q}}]$, every $\{x\}$-clause of $F_{\boldsymbol{q}}$ is *marked* as redundant. (A redundant clause is ignored by *DDS* until it is unmarked as non-redundant.) So, *DDS* terminates when the QE problem is solved for ξ in subspace $\boldsymbol{q}$.

B. Building Atomic D-sequents

Procedure *atomic_D_seqs* is called by *DDS* to compute D-sequents for trivial cases of variable redundancy listed in Subsection III-B. We refer to such D-sequents as **atomic**. Procedure *atomic_D_seqs* returns an updated set of active D-sequents Ω and answer *sat*, *unsat*, or *unknown* depending on whether F is satisfiable, unsatisfiable or its satisfiability is not known yet. If F is unsatisfiable, *atomic_D_seqs* also returns a clause C of F falsified by the current assignment $\boldsymbol{q}$.

Lines 1-3 of Figure 2 show what is done when F contains a clause C falsified by $\boldsymbol{q}$. In this case, every unassigned variable of F becomes redundant (Proposition 6). So, for every variable of $x \in X \setminus Vars(\boldsymbol{q})$ for which Ω does not contain a D-sequent yet, procedure *process_unsat_clause* generates D-sequent $\boldsymbol{g} \rightarrow \{x\}$ and adds it to Ω. Here $\boldsymbol{g}$ is the shortest assignment falsifying C. Once Ω contains a D-sequent for every variable of $X \setminus Vars(\boldsymbol{q})$, *atomic_D_seqs* terminates returning the answer *unsat*, set Ω and clause C.

Suppose no clause of F is falsified by $\boldsymbol{q}$. Then for every variable x of $X \setminus Vars(\boldsymbol{q})$ that does not have a D-sequent in Ω and that is blocked, a D-sequent is built as explained below. This D-sequent is then added to Ω (line 4). If every variable of $X \setminus Vars(\boldsymbol{q})$ has a D-sequent in Ω, then $F_{\boldsymbol{q}}$ is satisfiable. (If $F_{\boldsymbol{q}}$ is *unsatisfiable*, the variables of $X \setminus Vars(\boldsymbol{q})$ can be made redundant *only* by adding a clause falsified by $\boldsymbol{q}$.) So, *atomic_D_seqs* returns the answer *sat* and set Ω (line 5).

Given a blocked variable $x \in X \setminus \mathit{Vars}(\boldsymbol{q})$ of $F_{\boldsymbol{q}}$, a D-sequent $\boldsymbol{g} \rightarrow \{x\}$ is built as follows. Recall that an assignment $\boldsymbol{q}$ is a set of single-variable assignments. The fact that x is blocked in $F_{\boldsymbol{q}}$ means that for any pair of clauses C',C'' resolvable on x, C' or C'' is either satisfied by $\boldsymbol{q}$ or redundant (as containing a variable proved redundant in $\exists X[F_{\boldsymbol{q}}]$ earlier). Assume that it is clause C'. The assignment $\boldsymbol{g}$ is a subset of $\boldsymbol{q}$ guaranteeing that C' remains satisfied by $\boldsymbol{g}$ or redundant in $\exists X[F_{\boldsymbol{g}}]$ and so x remains blocked in $F_{\boldsymbol{g}}$. If C' is satisfied by $\boldsymbol{q}$, then $\boldsymbol{g}$ contains a single-variable assignment of $\boldsymbol{q}$ satisfying C'. If C' is not satisfied by $\boldsymbol{q}$ but contains a variable x^* proved redundant earlier, $\boldsymbol{g}$ contains all the single-variable assignments of $\boldsymbol{g^*}$ where $\boldsymbol{g^*} \rightarrow \{x^*\}$ is the D-sequent of Ω stating redundancy of x^*.

Searching for blocked variables of F for every call of DDS may be too expensive. Some simple techniques can be used to reduce the complexity of this search but a discussion of such techniques is beyond the scope of this paper. In the implementation of DDS we used in experiments, no optimization techniques were applied when searching for blocked variables.

C. Selection of a Branching Variable

Let $\boldsymbol{q}$ be the assignment DDS is called with and X_{red} be the set of variables of X whose D-sequents are in the current set Ω. Let $Y = \mathit{Vars}(F) \setminus X$. DDS branches only on a subset of free (*i.e.*, unassigned) variables of X and Y. Namely, a variable $x \in X \setminus \mathit{Vars}(\boldsymbol{q})$ is picked for branching only if $x \notin X_{red}$. A variable $y \in Y \setminus \mathit{Vars}(\boldsymbol{q})$ is picked for branching only if it is not detached. A variable y of $Y \setminus \mathit{Vars}(\boldsymbol{q})$ is called **detached** in $F_{\boldsymbol{q}}$, if every $\{y\}$-clause C of $F_{\boldsymbol{q}}$ that has at least one variable of X is redundant (because C contains a variable of X_{red}).

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 of C of $F_{\boldsymbol{q}}$ and v is picked for branching, then the value falsifying C is assigned first to cause immediate termination of this branch. In the description of DDS of Figure 1, the left branch always explores assignment $v = 0$. But, obviously, v can be first assigned value 1.

To simplify making the branching variable v redundant when merging results of the left and right branches (see Subsection V-E), *DDS first assigns values to variables of Y*. This means that *pick_variable* never selects a variable $x \in X$ for branching, if there is a free non-detached variable of Y. In particular, BCP does not assign values to variables of X if a non-detached variable of Y is still unassigned.

D. Switching from Left to Right Branch

DDS prunes big chunks of the search space by not branching on redundant variables of X or detached variables of Y. One more powerful pruning technique of DDS discussed in this subsection is to reduce the size of right branches.

Let $\boldsymbol{g} \rightarrow \{x\}$ be a D-sequent of the set Ω computed by DDS in the left branch $v = 0$ (line 5 of Figure 1).

```
merge($\xi, \boldsymbol{q}, v, \Omega^{asym}, \Omega$){
1  $\Omega := \mathit{join\_D\_seqs}(v, \Omega^{asym}, \Omega)$;
2  if ($v \in X$) $\Omega := \Omega \cup \{\mathit{atomic\_D\_seq\_for\_v}(F, \boldsymbol{q}, v, \Omega)\}$;
3  return($\Omega$);}
```

Fig. 3. *merge* procedure

Notice that if $\boldsymbol{g}$ has no assignment (v=0), variable x remains redundant in $\exists X[F_{\boldsymbol{q_1}}]$ where $\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$. This is because $\boldsymbol{g} \rightarrow \{x\}$ is still active in the subspace specified by $\boldsymbol{q_1}$. DDS splits the set Ω into subsets Ω^{sym} and Ω^{asym} of D-sequents symmetric and asymmetric with respect to variable v (line 6). We call a D-sequent $\boldsymbol{g} \rightarrow \{x\}$ *symmetric* with respect to v, if $\boldsymbol{g}$ does not contain an assignment to v and *asymmetric* otherwise.

Denote by X^{sym} and X^{asym} the variables of $X_{red} \setminus \mathit{Vars}(\boldsymbol{q})$ whose redundancy is stated by D-sequents of Ω^{sym} and Ω^{asym} respectively. Before exploring the right branch (line 9), the variables of X^{asym} become non-redundant again. Every clause C of $F_{\boldsymbol{q}}$ with a variable of X^{asym} is unmarked as currently non-redundant unless $\mathit{Vars}(C) \cap X^{sym} \neq \emptyset$.

Reducing the set of free variables of the right branch to X^{asym} allows to prune big parts of the search space. In particular, if X^{asym} is empty there is no need to explore the right branch. In this case, DDS just returns the results of the left branch (line 7). Pruning the right branch when X^{asym} is empty is similar to non-chronological backtracking well known in SAT-solving [21].

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 extend the redundancy of all unassigned variables of X proved in $\exists X[F_{\boldsymbol{q_0}}]$ and $\exists X[F_{\boldsymbol{q_1}}]$ to formula $\exists X[F_{\boldsymbol{q}}]$. If both $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ turned out to be unsatisfiable, this is done as described in lines 11-14 of Figure 1. In this case, the unsatisfied clauses C_0 and C_1 of $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ returned in the left and right branches respectively are resolved on v. The resolvent C is added to F. Since F contains a clause C that is falsified by $\boldsymbol{q}$, for every variable $x \in X \setminus \mathit{Vars}(\boldsymbol{q})$ whose D-sequent is not in Ω, DDS derives an atomic D-sequent and adds it to Ω. This is performed by procedure *process_unsat_clause* described in Subsection V-B. If, say, $v \notin \mathit{Vars}(C_1)$, then *resolve_clauses* (line 11) returns C_1 itself since C_1 is falsified by $\boldsymbol{q}$ and no new clause is added to F.

If at least one branch returns answer *sat*, then DDS calls procedure *merge* described in Figure 3. First, *merge* takes care of the variables of X^{asym} (see Subsection V-D). Note that redundancy of variables of X^{asym} is already proved in both branches. If a D-sequent of a variable from X^{asym} returned in the *right* branch is asymmetric in v, then *join_D_seqs* (line 1) replaces it with a D-sequent symmetric in v as follows.

Let $x \in X^{asym}$ and S_0 and S_1 be the D-sequents stating the redundancy of x derived in the left and right branches respectively. Then *join_D_seqs* joins S_0 and S_1 at v producing a new D-sequent S. The latter also states the redundancy of x but does not depend on v. D-sequent S_1 is replaced in Ω with S. If S_1 itself does not depend on v, no new D-sequent is produced. S_1 remains in Ω as the active D-sequent for variable x in $F_{\boldsymbol{q}}$.

Finally, if the branching variable v is in X, DDS derives a D-sequent stating the redundancy of v. Notice that v is not currently redundant in $\exists X[F_{\boldsymbol{q}}]$ because DDS does not branch on redundant variables. As we mentioned in Subsection V-C, the variables of $Y = \mathit{Vars}(F) \setminus X$ are assigned in DDS before those of X. This means that before v was selected for branching, all free non-detached variables of Y had been assigned. Besides, every variable of $X \setminus \mathit{Vars}(\boldsymbol{q})$ but v has just been proved redundant in $\exists X[F_{\boldsymbol{q}}]$. So, $F_{\boldsymbol{q}}$ may have only two types of non-redundant clauses: a) clauses having only detached variables of Y; b) unit clauses depending on v. Moreover, these unit clauses cannot contain literals of both polarities of v because *merge* is called only when either branch $v = 0$ or $v = 1$ is satisfied. Therefore, v is monotone. An atomic D-sequent S stating the redundancy of v is built as described in Subsection V-B and added to Ω (line 2). Then *merge* terminates returning Ω.

F. Correctness of DDS and Example

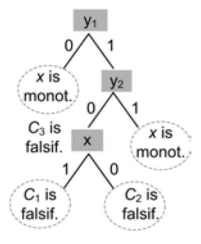


Fig. 4. Search tree built by DDS

Let DDS be called on formula $\xi = \exists X[F]$ with $\boldsymbol{q} = \emptyset$ and $\Omega = \emptyset$. Informally, DDS is correct because a) the atomic D-sequents built by DDS are correct; b) joining D-sequents produces a correct D-sequent; c) every clause added to formula F is produced by resolution and so is implied by F; d) by the time DDS backtracks to the root of the search tree, for every variable $x \in X$, D-sequent $\emptyset \rightarrow \{x\}$ is derived. Due to Proposition 4, this implies that the D-sequent $\emptyset \rightarrow X$ holds for the formula $\exists X[F]$ returned by DDS.

Proposition 9: DDS is sound and complete.

As we mentioned earlier, the proofs of the propositions given in this paper are provided in [16].

Example 2. 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 DDS call we will use the corresponding assignment $\boldsymbol{q}$. For example, $DDS_{(y_1=1,y_2=0)}$ means that the assignments $y_1 = 1$ and $y_2 = 0$ were made at recursion depths 0 and 1 respectively. So the current recursion depth is 2. Originally, assignment $\boldsymbol{q}$ is empty so the initial call is $DDS_{(\emptyset)}$. The work of DDS is shown in Figures 4 and 5 used below to illustrate various aspects of DDS.

Branching variables. Figure 4 shows a search tree built by DDS. Recall that DDS branches on variables of $\mathit{Vars}(F) \setminus X = \{y_1, y_2\}$ 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, variable x is either assigned or proved redundant. For example, x is proved redundant by $DDS_{(y_1=0)}$ and assigned by $DDS_{(y_1=1,y_2=0,x=1)}$.

Generation of new clauses. $DDS_{(y_1=1,y_2=0)}$ generates a new clause after branching on x. $DDS_{(y_1=1,y_2=0,x=1)}$ returns C_1 as a clause of F that is empty in $F_{(y_1=1,y_2=0,x=1)}$. Similarly, $DDS_{(y_1=1,y_2=0,x=0)}$ returns C_2 because it is empty in $F_{(y_1=1,y_2=0,x=0)}$. As described in Subsection V-E, in this case, DDS 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.

Generation of atomic D-sequents. Figure 5 describes derivation of D-sequents. The dotted boxes show D-sequents obtained by the join operation. The atomic D-sequents are shown in dotted ovals. For instance, $DDS_{(y_1=0)}$ generates D-sequent S_1 equal to $(y_1 = 0) \rightarrow \{x\}$. S_1 holds because $F_{(y_1=0)}$=$y_2 \vee x$ and so x is a blocked (monotone) variable of $F_{(y_1=0)}$. The atomic D-sequent S_2 is derived by $DDS_{(y_1=1,y_2=0)}$. As we mentioned above, $DDS_{(y_1=1,y_2=0)}$ adds clause $C_3 = \overline{y}_1 \vee y_2$ to F. This clause is empty in $F_{(y_1=1,y_2=0)}$. So D-sequent S_2 equal to $(y_1 = 1, y_2 = 0) \rightarrow \{x\}$ is generated where $(y_1 = 1, y_2 = 0)$ is the shortest assignment falsifying C_3.

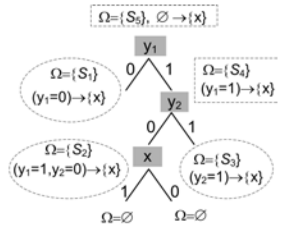


Fig. 5. Derivation of D-sequents

Switching from left to right branch. Let us consider switching between branches by $DDS_{(\emptyset)}$ where y_1 is picked for branching. The set of D-sequents $\Omega_{(\emptyset)}$ returned by the left branch equals $\{S_1\}$ where S_1 is equal to $(y_1 = 0) \rightarrow \{x\}$. The only clause $y_2 \vee x$ of $F_{(y_1=0)}$ is marked as redundant because it contains x that is currently redundant. Before starting the right branch $y_1 = 1$, $DDS_{(\emptyset)}$ splits $\Omega_{(\emptyset)}$ into subsets $\Omega^{sym}_{(\emptyset)}$ and $\Omega^{asym}_{(\emptyset)}$ of D-sequents respectively symmetric and asymmetric in y_1. Since the only D-sequent of $\Omega_{(\emptyset)}$ depends on y_1, then $\Omega^{asym}_{(\emptyset)}$=$\Omega_{(\emptyset)}$ and $\Omega^{sym}_{(\emptyset)}$=$\emptyset$. $DDS_{(\emptyset)}$ removes D-sequent S_1 from Ω because S_1 is not active if $y_1 = 1$. So, before $DDS_{(y_1=1)}$ is called, variable x becomes non-redundant and clause $C_2 = y_2 \vee x$ is unmarked as currently non-redundant.

Branch merging. Consider how branch merging is performed by $DDS_{(y_1=1)}$. In the left branch $y_2 = 0$, the set $\Omega_{(y_1=1)}$=$\{S_2\}$ is computed where S_2 is $(y_1 = 1, y_2 = 0) \rightarrow \{x\}$. Since S_2 depends on y_2, then $\Omega^{asym}_{(y_1=1)}$=$\Omega_{(y_1=1)}$. In the right branch $y_2 = 1$, the set $\Omega_{(y_1=1)}$=$\{S_3\}$ is computed where S_3 is $(y_2 = 1) \rightarrow \{x\}$. By joining S_2 and S_3 at y_2, D-sequent S_4 is derived that equals $(y_1 = 1) \rightarrow \{x\}$. S_4 states redundancy of x in $F_{(y_1=1)}$.

Termination. When $DDS_{(\emptyset)}$ terminates, $F = C_1 \wedge C_2 \wedge C_3$ where $C_3 = \overline{y}_1 \vee y_2$ and D-sequent $\emptyset \rightarrow \{x\}$ is derived. By dropping C_1, C_2 as X-clauses one obtains $C_3 \equiv \exists X[C_1 \wedge C_2]$.

VI. Compositionality of DDS

We will call a CNF formula F **compositional** if $F = F_1 \wedge \ldots \wedge F_k$ where $\mathit{Vars}(F_i) \cap \mathit{Vars}(F_j) = \emptyset$, $i \neq j$. We will say that an algorithm solves the QE problem specified by $\exists X[F]$ **compositionally** if it breaks this problem down into k independent subproblems of finding G_i equivalent to $\exists X[F_i]$. A formula G equivalent to $\exists X[F]$ is then built as $G_1 \wedge \ldots \wedge G_k$.

There are at least two reasons to look for compositional QE algorithms. First, even if the *original* formula F is not compositional, a formula $F_{\boldsymbol{q}}$ obtained from F by making assignment $\boldsymbol{q}$ may be compositional. Second, a practical formula F typically can be represented as $F_1(X_1,Y_1) \wedge \ldots \wedge F_k(X_k,Y_k)$ where X_i are internal variables of F_i and Y_i are communication variables i.e. ones shared by subformulas F_i. One can view F_i as describing a "design block" with external variables Y_i. The size of Y_i is usually much smaller than that of X_i. The latter fact is, arguably, what one means by saying that F has structure. One can view compositional formulas as a degenerate case where $|Y_i| = 0, i = 1,\ldots,k$ and so F_i do not "talk" to each other. Intuitively, an algorithm that does not scale well even if $|Y_i| = 0$ will not do well when $|Y_i| > 0$.

A QE algorithm based on enumeration of satisfying assignments is not compositional. The reason is that the set of assignments satisfying F is a Cartesian product of those satisfying F_i,$i = 1,\ldots,k$. So if, for example, all F_i are identical, the complexity of an enumeration based QE algorithm is *exponential* in k. A QE algorithm based on BDDs [7] is compositional only for variable orderings where variables of F_i and F_j, $i \neq j$ do not interleave.

Now we show the compositionality of DDS. By a *decision branching variable* mentioned in the proposition below, we mean that this variable was not present in a unit clause of the current formula when it was selected for branching.

Proposition 10 (compositionality of DDS): Let T be the search tree built by DDS when solving the QE problem $\exists X[F_1 \wedge \ldots \wedge F_k]$ above. Let $X_i = X \cap \mathit{Vars}(F_i)$ and $Y_i = \mathit{Vars}(F_i) \setminus X$. The size of T in the number of nodes is bounded by $|\mathit{Vars}(F)| \cdot (\eta(X_1 \cup Y_1) + \ldots + \eta(X_k \cup Y_k))$ where $\eta(X_i \cup Y_i) = 2 \cdot 3^{|X_i \cup Y_i|} \cdot (|X_i| + 1), i = 1,\ldots,k$ no matter how decision branching variables are chosen.

Proposition 10 is proved in [16] for a slightly modified version of DDS. Notice that the compositionality of DDS is not ideal. For example, if all subformulas F_i are identical, DDS is *quadratic* in k as opposed to being linear. Informally, DDS is compositional because D-sequents it derives have the form $\boldsymbol{g} \rightarrow \{x\}$ where $\mathit{Vars}(\boldsymbol{g}) \cup \{x\} \subseteq \mathit{Vars}(F_i)$. The only exception are D-sequents derived when the current assignment falsifies a clause of F. This exception is the reason why DDS is quadratic in k.

Importantly, the compositionality of DDS is achieved not by using some ad hoc techniques but is simply a result of applying the machinery of D-sequents. This provides some evidence that DDS can be successfully applied to non-compositional formulas of the form $F_1(X_1,Y_1) \wedge \ldots \wedge F_k(X_k,Y_k)$ where $|Y_i| > 0$ and $|Y_i| \ll |X_i|$, $i = 1,\ldots,k$.

Notice that a QE algorithm that resolves out variables one by one as in the DP procedure [12] is also compositional. (If $\mathit{Vars}(F_i) \cap \mathit{Vars}(F_j) = \emptyset$, then clauses of F_i and F_j cannot be resolved with each other). However, although such an algorithm may perform well on some classes of formulas, it is not very promising overall. This is due to the necessity to eliminate a variable in one big step, which may lead to generation of a very large number of new resolvent clauses. On the contrary, being a branching algorithm, DDS is very opportunistic and eliminates the same variable differently in different subspaces trying to reduce the number of new resolvents to be added (if any). The lack of flexibility in variable elimination is exactly the cause of the poor scalability of the DP procedure in SAT-solving. There is no reason to believe that DP-like procedures will scale better for the harder problem of quantifier elimination.

As we mentioned above, QE algorithms based on BDDs are compositional only for particular variable orders. This limitation coupled with the necessity for a BDD to maintain one global variable order may cripple the performance of BDD based algorithms even on very simple formulas. Suppose, for instance, that H and G are compositional CNF formulas where $H = H_1 \wedge \ldots \wedge H_k$ and $G = G_1 \wedge \ldots \wedge G_m$. Suppose that variables of subformulas of H and G overlap with each other so that every variable order for which a BDD of G is small renders a large BDD for H and vice versa. Let F be a CNF formula equivalent to $(w \vee H) \wedge (\overline{w} \vee G)$ where $w \notin \mathit{Vars}(H) \cup \mathit{Vars}(G)$. (A CNF formula for, say, $w \vee H$ is trivially obtained by adding literal w to every clause of H.) Notice that F is compositional in branches $w = 0$ and $w = 1$ since $F_{w=0} = H$ and $F_{w=1} = G$. However, a BDD based QE algorithm cannot benefit from this fact because the same variable order has to be used in either branch and no order is good for both H and G. Notice, that DDS will not have any problem in handling formula F because DDS is compositional for any choice of decision variables in branches $w = 0$ and $w = 1$.

VII. Experimental Results

The objective of experiments was to compare DDS with other SAT-based QE algorithms. We are planning to make a comparison of DDS with BDD-based algorithms in the near future. In our experiments, we used a QE algorithm based on enumeration of satisfying assignments [6] (courtesy of Andy King). We will refer to this QE algorithm as *EnumSA*. We also compared DDS with the QE algorithm of [15] that we will call *QE-GBL*. Here *GBL* stands for *global*. Given a formula $\exists X[F]$, *QE-GBL* eliminates variables of X globally, one by one, as in the DP procedure. However, when resolving out a variable $x \in X$, *QE-GBL* adds a new resolvent to F *only if* it eliminates an $\{x\}$-removable $\{x\}$-boundary point of F. Variable x is redundant in $\exists x[F]$ if all $\{x\}$-removable $\{x\}$-boundary points of F are eliminated. *QE-GBL* does not generate so many redundant clauses as DP, but still has the flaw of eliminating variables globally.

We used *QE-GBL* for two reasons. First, DDS can be viewed as a branching version of *QE-GBL*. In Section VI, we argued that branching gives DDS more flexibility in variable elimination in comparison to procedures eliminating variables globally. So we wanted to confirm that DDS indeed benefited from branching. Second, one can consider *QE-GBL* as an algorithm similar to that of [18]. The latter solves $\exists x[F(x,Y)]$ by looking for a Boolean function $H(Y)$ such that $F(H(Y),Y) \equiv \exists x[F(x,Y)]$. We used *QE-GBL* to get an idea about the performance of the algorithm of [18] since it was not implemented as a stand-alone tool.Our implementation of *QE-GBL* was quite efficient. In particular, we employed Picosat [4] for finding boundary points.

TABLE I
Experiments with model checking formulas. The time limit is 1min

model checking mode	*EnumSA*		*QE-GBL*		*DDS*	
	solved (%)	time (s.)	solved (%)	time (s.)	solved (%)	time (s.)
forward	425 (56%)	466	561 (74%)	4,865	664 (87%)	1,530
backward	97 (12%)	143	522 (68%)	2,744	563 (74%)	554

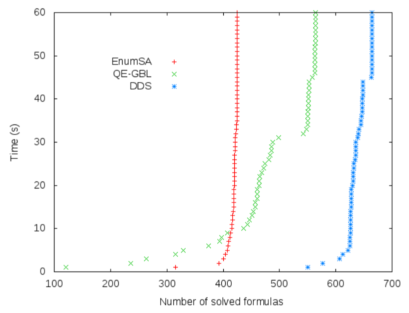


Fig. 6. Forward model checking (1 iteration)

In the first two experiments (Table I), we used the 758 model checking benchmarks of HWMCC'10 competition [27]. In the first experiment (the first line of Table I) we used *EnumSA*, *QE-GBL* and *DDS* to compute the set of states S^1_{reach} reachable in the first transition. In this case, CNF formula F describes the transition relation and the initial state. CNF formula G equivalent to $\exists X[F]$ specifies S^1_{reach}.

In the second experiment, (the second line of Table I) we used the same benchmarks to compute the set of "bad" states in backward model checking. In this case, F specifies the output function and the property in question. If F evaluates to 1 for some assignment $\boldsymbol{p}$ to $\mathit{Vars}(F)$, this property is broken and the state given by the state bits of $\boldsymbol{p}$ is bad. Formula G equivalent to $\exists X[F]$ specifies the set of all bad states (that may or may not be reachable from the initial state).

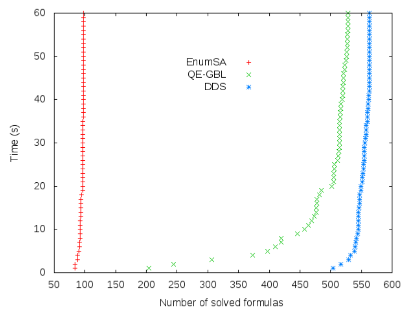


Fig. 7. Backward model checking (1 iteration)

Table I shows the comparison of the three programs with respect to the number of formulas solved, percentage of this number to the total number (758) and time taken for the *solved* problems. With 1-minute time limit, *DDS* solved more formulas than *EnumSA* and *QE-GBL* in forward and backward model checking. Figures 6 and 7 give the number of formulas of Table I solved by the three programs in t seconds, $0 \leq t \leq 60$. These figures show the superiority of DDS over *QE-GBL* and *EnumSA* on the set of formulas we used. The poor performance of *EnumSA* on backward model checking formulas is due to lack of constraints on next state variables. In the presence of such constraints, *EnumSA* performs much better (see below).

The size of the 1,227 formulas solved by *DDS* peaked at 98,105 variables, the medium size being 2,247 variables. The largest number of non-quantified (*i.e.*, state) variables was 7,880 and 541 formulas had more than 100 state variables. The size of resulting formula G peaked at 32,769 clauses, 361 resulting formulas had more than 100 clauses. We used Picosat [4] to remove redundant literals and clauses of G. Namely, for every clause C of G we checked if G was equivalent to $G \setminus \{C\}$. If so, C was removed from G. Otherwise, we tested every literal l of C if removal l from C changed the function of G. If not, l was removed from C. The total runtime for the optimization of G by Picosat was limited by 4 seconds. Overall, the resulting formulas built by *DDS* were smaller than those of *EnumSA* and *QE-GBL*. For instance, out of 1069 formulas solved by both *DDS* and *QE-GBL*, the size of G built by *DDS* was smaller (respectively equal or larger) in 267 (respectively 798 and 4) cases.

TABLE II
Compositionality of QE algorithms. Time limit=1hour

#copies	#vars, #clauses	$\|Y\|$	*EnumSA* (s.)	*DDS* rand (s.)	*DDS* (s.)
5	20,30	10	0	0.01	0.01
10	40,60	20	10.5	0.01	0.01
15	60,90	30	>1hour	0.01	0.01
500	2000,3000	1000	>1hour	1.95	0.04

In the experiments above, we did not use formula preprocessing even though it could have been beneficial. For instance, the forward model checking formulas had a lot of unit clauses encoding the initial state. The backward model checking formulas had many blocked (*i.e.*, redundant) clauses [5]. The reason is that when the *original* set of bad states is computed, the next state variables are not constrained yet. However, when we compared the three programs on preprocessed formulas we obtained similar results: *DDS* outperformed *EnumSA* and *QE-GBL*. In particular, we generated 189 backward model checking formulas specifying bad states after a number of iterations. The idea was to get formulas were preprocessing simplifications performing initial BCP and elimination of blocked clauses failed. With 1-minute time limit, *DDS*, *QE-GBL* and *EnumSA* solved 185, 163 and 149 formulas out of 189 respectively. Notice that *EnumSA* performed much better here than in the initial iteration.

The third experiment (Table II), clearly shows the compositionality of *DDS* in comparison to *EnumSA*. In this experiment, both programs computed the output assignments produced by a combinational circuit N composed of small *identical* circuits $N_1, \ldots, N_k$ with independent sets of variables. In this case, one needs to eliminate quantifiers from $\exists X[F]$ where $F = F_1 \wedge \ldots \wedge F_k$. CNF formula F_i specifies N_i and $\mathit{Vars}(F_i) \setminus X$ and $\mathit{Vars}(F_i) \cap X$ are the sets of output and non-output variables of N_i respectively. So a CNF formula equivalent to $\exists X[F]$ specifies the output assignments of N.

The first column of Table II shows k (the number of copies of N_i). The next two columns give the size of CNF formula F and the number of outputs in circuit N. The last three columns show the run time of *EnumSA* and two versions of DDS. In the first version, the choice of branching variables was random. In the second version, this choice was guided by the compositional structure of N. While DDS solved all the formulas easily, *EnumSA* could not finish the formulas F with $k \geq 15$ in 1 hour. Notice that DDS was able to quickly solve all the formulas even with the random choice of branching variables.

VIII. Background

The relation between a resolution proof and the process of elimination of boundary points was discussed in [14]. In terms of the present paper, [14] dealt only with a special kind of Z-boundary points of formula F where $|Z| = 1$. In the present paper, we consider the case where Z is an arbitrary subset of the set of quantified variables X of an $\exists$CNF formula $\exists X[F]$. This extension is crucial for describing the semantics of D-sequents.

The notion of D-sequents was introduced in [17]. There, we formulated a QE algorithm that branched only on quantified variables of $\exists X[F]$. This algorithm is more complex than DDS because it has to compute boundary points explicitly. At the same time, as we mentioned in the introduction, the limitation on variable order used by DDS (see Subsection V-C) is artificial. In general, to achieve the best results one has to interleave assignments to quantified and non-quantified variables. Then to reduce the number of resolvent clauses to be added one needs to compute boundary points explicitly.

As far as quantifier elimination is concerned, QE algorithms and QBF solvers can be partitioned into two categories. (Although, in contrast to a QE algorithm, a QBF-solver is a decision procedure, they both employ methods of quantifier elimination. Since this paper is focused on SAT-based solvers, we omit references to papers on QE algorithms that use BDDs [7], [8].) The members of the first category employ various techniques to eliminate quantified variables of the formula one by one in some order [26], [3], [2], [18], [1]. For example, in [18], quantified variables are eliminated by interpolation. All these solvers face the problem that we already discussed in Section VI. The necessity to eliminate a variable in one big step deprives the algorithm of flexibility and, in general, leads to generation of prohibitively large sets of clauses.

The solvers of the second category are based on enumeration of satisfying or unsatisfying assignments [23], [19], [13], [6], [25]. 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. 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.

IX. Conclusion

We introduced Derivation of Dependency-sequents (DDS), a new method for eliminating quantifiers from a formula $\exists X[F]$ where F is a CNF formula. The essence of DDS is to add resolvent clauses to F to make the variables of X redundant. The process of making variables redundant is described by dependency sequents (D-sequents) specifying conditions under which variables of X are redundant. In contrast to methods based on the enumeration of satisfying assignments, DDS is compositional. Our experiments with a proof-of-the-concept implementation show the promise of DDS. Our future work will focus on studying various ways to improve the performance of DDS, including lifting the constraint that non-quantified variables are assigned before quantified variables and reusing D-sequents instead of discarding them after one join operation (as SAT-solvers reuse conflict clauses).

Acknowledgment

This work was funded in part by NSF grant CCF-1117184, NASA NASA Cooperative Agreement NNX08AE37A, and DARPA under Air Force Research Laboratory (AFRL/Rome) Cooperative Agreement No. FA8750-10-2-0233.

References

[1] P. Abdulla, P. Bjesse, and N. Eén, "Symbolic reachability analysis based on SAT-solvers," in *Proc. of TACAS*, 2000, pp. 411–425.

[2] A. Ayari and D. Basin, "Qubos: Deciding quantified boolean logic using propositional satisfiability solvers," in *FMCAD*, 2002, pp. 187–201.

[3] A. Biere, "Resolve and expand," in *Proc. of SAT-04*, 2005, pp. 59–70.

[4] ——, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[5] A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for qbf," in *Proc. of CADE*, 2011, pp. 101–115.

[6] J. Brauer, A. King, and J. Kriener, "Existential quantificationnn as incremental sat," in *Proc. of CAV-11*. Springer-Verlag, July 2011, pp. 191–207.

[7] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[8] P. Chauhan, E. Clarke., S. Jha, J. Kukula, H. Veith, and D. Wang, "Using combinatorial optimization methods for quantification scheduling," in *Proc. of CHARME*, 2001, pp. 293–309.

[9] E. Clarke and A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*, 1982, pp. 52–71.

[10] E. Clarke, O. Grumberg, and D. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[11] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, July 1962.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, July 1960.

[13] M. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based unbounded symbolic model checking using circuit cofactoring," in *Proc. of ICCAD*, 2004, pp. 510–517.

[14] E. Goldberg, "Boundary points and resolution," in *Proc. of SAT*. Springer-Verlag, 2009, pp. 147–160.

[15] E. Goldberg and P. Manolios, "Sat-solving based on boundary point elimination," in *Proc. of HVC-10*. Springer-Verlag, 2011, pp. 93–111.

[16] ——, "Quantifier elimination by dependency sequents," Northeastern University, Tech. Rep. arXiv:1201.5653v3 [cs.LO], 2012. [Online]. Available: http://arxiv.org/pdf/1201.5653v3

[17] ——, “Removal of quantifiers by elimination of boundary points,” Northeastern University, Tech. Rep. arXiv:1204.1746v2 [cs.LO], 2012. [Online]. Available: http://arxiv.org/pdf/1204.1746v2

[18] R. Jiang, “Quantifier elimination via functional composition,” in *Proc. of CAV ’09*. Springer, 2009, pp. 383–397.

[19] H. Jin and F. Somenzi, “Prime clauses for fast enumeration of satisfying assignments to boolean circuits,” in *Proc. of DAC*, 2005, pp. 750–753.

[20] O. Kullmann, “New methods for 3-sat decision and worst-case analysis,” *Theor. Comput. Sci.*, vol. 223, no. 1-2, pp. 1–72, Jul. 1999.

[21] J. Marques-Silva and K. Sakallah, “Grasp—a new search algorithm for satisfiability,” in *ICCAD-96*, Washington, DC, USA, 1996, pp. 220–227.

[22] K. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[23] ——, “Applying sat methods in unbounded symbolic model checking,” in *Proc. of CAV-02*. Springer-Verlag, 2002, pp. 250–264.

[24] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: engineering an efficient sat solver,” in *DAC-01*, New York, NY, USA, 2001, pp. 530–535.

[25] D. Plaisted, A. Biere, and Y. Zhu, “A satisfiability procedure for quantified boolean formulae,” *Discrete Appl. Math.*, vol. 130, no. 2, pp. 291–328, Aug. 2003.

[26] P. Williams, A. Biere, E. Clarke, and A. Gupta, “Combining decision diagrams and sat procedures for efficient symbolic model checking,” in *Proc. of CAV*, 2000, pp. 124–138.

[27] HWMCC-2010 benchmarks, http://fmv.jku.at/hwmcc10/benchmarks.html.

Preprocessing Techniques for First-Order Clausification

Krystof Hoder
Computer Science Department
University of Manchester, UK
hoderk@cs.man.ac.uk

Zurab Khasidashvili
Intel Israel (74) Ltd.
Haifa 31015, Israel
zurabk@iil.intel.com

Konstantin Korovin, Andrei Voronkov
Computer Science Department
University of Manchester, UK
korovin@cs.man.ac.uk, andrei@voronkov.com

***Abstract*—It is well known that preprocessing is crucial for efficient reasoning on large industrial problems. Although preprocessing is well developed for propositional logic, it is much less investigated for first-order logic. In this paper we introduce several preprocessing techniques for simplifying first-order formulas aimed at improving clausification. These include definition inlining and merging, simplifications based on a new data structure, quantified AIG, and its combination with BDDs. We implemented our preprocessing methods and evaluated them over encodings of industrial hardware verification problems into the effectively propositional (EPR) fragment of first-order logic and over standard first-order (TPTP) and SMT (SMT-LIB) benchmarks. We also investigated preprocessing methods that help obtain EPR-resulting clausification in cases where standard clausification would lead outside the EPR fragment. We demonstrate that our methods enable one to considerably reduce the number of clauses obtained after classification and by that help speedup first-order reasoning.**

I. Introduction

First-order logic solvers are increasingly used in industrial verification applications. These uses include model checking of large real-life hardware systems. It is well known that hardware designs have many redundancies from the logical point of view. Many powerful techniques have been developed for propositional logic problems to eliminate these redundancies. These techniques include use of efficient representations for propositional formulas, such as AIGs (And-Inverter Graphs) [17], simplification transformations for AIGs, such as BDD-sweeping, SAT-sweeping, AIG-rewriting [17], [16], [5], [7], and various pre- and in-processing techniques, e.g., [12], [10] which aim to simplify propositional problems for SAT and QBF solving. In this work, motivated by attempts to improve the performance and capacity of a model-checking algorithm we have recently developed [9], we seek to develop general simplification techniques for first-order logic problems.

First-order definitions are frequently used in many formalizations. For example, in hardware verification most generated formulas are definitions. An abundance of definitions can considerably slowdown the reasoning process. Many definitions in such problems are redundant, defining equivalent formulas, or can be eliminated without increasing the formula size. Moreover, direct clausal transformation of definitions can lead outside target fragments such as the effectively propositional (EPR) fragment (see definition in the next section). In this paper we introduce and discuss several methods for eliminating and simplifying definitions that also result in EPR-preserving clausification.

We further lift some of the propositional redundancy elimination techniques discussed above to first-order logic. In particular, we introduce quantified AIGs as an efficient data structure that enables sharing equivalent sub-formulas and facilitates implementation of simplification transformations for first-order logic formulas. On QAIGs, we implement BDD-sweeping, SAT-sweeping, and several rewriting transformations that help reduce the size of the problem after clausification and thus making the problem much simpler to solve.

Our improved clausification algorithm (which, as preprocessing steps, performs the above mentioned simplification transformations) is implemented in Vampire, a theorem prover for first-order logic [11]. We have evaluated the new clausification algorithm on three different benchmark sets: industrial hardware designs, quantified SMT problems and a TPTP problem set [22]. The experiments demonstrate the usefulness of our simplification transformations.

The paper is organized as follows. In Section II we recall basic definitions from first-order logic used throughout the paper. Sections III to XI are devoted to a range of simplification techniques for first-order logic formulas. Quantified AIGs are introduced and studied in Section XII. Experimental results are reported in Section XIII. Conclusions appear in Section XIV.

II. Preliminaries

We say that a formula φ is rectified if the following holds: (i) no variable occurs both free and bound in φ, and (ii) a variable can have at most one binding occurrence in φ. For simplicity of exposition, we assume that our formulas are rectified unless otherwise specified. In particular, this requirement is dropped in sections concerned with shared representation of formulas, such as AIGs and OBBDs, in order to increase sharing between subformulas.

We consider first-order formulas which are built from atoms using connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and quantifiers $\exists$ and $\forall$. We assume the standard semantics of first-order formulas. Polarity of a subformula occurrence at a position π will be denoted by $pol(\varphi, \pi) \in \{-1, 0, 1\}$, where 1 stands for positive, -1 for negative, and 0 for neutral polarity, which is defined inductively as follows. For any formula φ, $pol(\varphi, \epsilon) = 1$.

Consider $\varphi\mid_\pi = \psi$ and assume $pol(\varphi,\pi)$ is defined, then if ψ is of the form

- $Qx\ \psi_1$, where $Q \in \{\exists, \forall\}$ then $pol(\varphi,\pi.1) = pol(\varphi,\pi)$;
- $\psi_1 \star \psi_2$, where $\star \in \{\wedge,\vee\}$ then $pol(\varphi,\pi.1) = pol(\varphi,\pi.2) = pol(\varphi,\pi)$;
- $\psi_1 \rightarrow \psi_2$ then $pol(\varphi,\pi.1) = -pol(\varphi,\pi)$ and $pol(\varphi,\pi.2) = pol(\varphi,\pi)$;
- $\psi_1 \leftrightarrow \psi_2$ then $pol(\varphi,\pi.1) = pol(\varphi,\pi.2) = 0$.

Algorithms in this paper are parameterized by a Skolemization procedure $\mathcal{SK}$ and a clausification procedure $\mathcal{CL}$. In this paper we are not concerned how $\mathcal{SK}$ is realized, assuming only that $\mathcal{SK}$ transforms every first-order formula into an equi-satisfiable universal formula. We refer to [20], [2] for Skolemization and clausification techniques. As an example, we take an $\mathcal{SK}$ that applies miniscoping (moving all quantifiers inside the formula as far as possible) and eliminates existential quantifiers, as in inner Skolemization from left-to-right (resulting in flat Skolem terms). Similarly, we only require the clausification $\mathcal{CL}$ to transform universal formulas into equi-satisfiable sets of clauses.

The EPR fragment, also called the Bernays-Schönfinkel-Ramsey fragment, consists of first-order formulas with no occurrences of function symbols other than constants, and which when written in prenex normal form have the quantifier prefix $\exists^*\forall^*$. Skolemization applied to EPR formulas can introduce only constant function symbols; this can be used to show decidability of the EPR fragment. Several important verification problems have been encoded into EPR [19], [13], [8], [9], [1], benefiting from the succinct representations possible in this fragment. The transformations considered in this paper can help to produce equi-satisfiable EPR formulas when the given formula is not necessarily EPR. Such transformations turned out to be crucial for the performance of first-order solvers on encodings of real-life hardware verification problems.

III. Definition simplifications

A (non-recursive) *predicate definition* $def(pol, p, \varphi)$ is a first-order formula of the form

$$def(pol,p,\varphi) \overset{\text{def}}{=} \begin{cases} \forall\bar{x}\ (p(\bar{x}) \leftrightarrow \varphi(\bar{x})), & \text{if } pol = 0,\\ \forall\bar{x}\ (\varphi(\bar{x}) \rightarrow p(\bar{x})), & \text{if } pol = 1,\\ \forall\bar{x}\ (p(\bar{x}) \rightarrow \varphi(\bar{x})), & \text{if } pol = -1, \end{cases} \tag{1}$$

where p is a predicate symbol, φ is a first-order formula with free variables $FV(\varphi) \subseteq \{\bar{x}\}$, $pol \in \{-1,0,1\}$ and p does not occur in φ. Let us note that $def(0,p,\varphi) \equiv (def(1,p,\varphi) \wedge def(-1,p,\varphi))$; we call $def(1,p,\varphi)$ *positive* and $def(-1,p,\varphi)$ *negative* subdefinition of $def(0,p,\varphi)$. The variable condition $FV(\varphi) \subseteq \{\bar{x}\}$ can be omitted without loss of generality but doing so would add a syntactic burden not essential to this exposition.

First we consider unused definition elimination, presented in Table I.

Theorem 1: UDE is a satisfiability preserving and terminating transformation.

$\varphi \wedge def(pol,p,\psi)$	$\Rightarrow$	φ, where p does not occur in φ.
$\varphi \wedge def(0,p,\psi)$	$\Rightarrow$	$\varphi \wedge def(pol,p,\psi)$, where $pol \neq 0$ and all occurrences of p in φ are of polarity $-pol$.

TABLE I
Unused Definition Elimination (UDE)

Proof: (Sketch) Termination is trivial since each application removes one (sub)definition. Let us show that UDE is satisfiability preserving. Consider the case $\varphi \wedge def(0,p,\psi) \Rightarrow \varphi \wedge def(-1,p,\psi)$, where all occurrences of p in φ are of polarity 1. The rest of the cases are similar. The only non-trivial direction is to show that if $\varphi \wedge def(-1,p,\psi)$ is satisfiable then $\varphi \wedge def(0,p,\psi)$ is also satisfiable. First note that if a predicate occurs only positively in a formula $\chi(\bar{x})$ then the formula is monotone wrt. this predicate in the following sense. Consider a first-order interpretation I such that $I \models \chi(\bar{a})$. Then $I' \models \chi(\bar{a})$ for any I' which is obtained from I by changing the interpretation of p such that $p^I \subseteq p^{I'}$. Now assume that $\varphi \wedge def(-1,p,\psi)$ is satisfiable in a model I and p occurs only positively in φ. Let I' be obtained from I by changing the interpretation of p such that $I' \models p(\bar{a})$ iff $I \models \psi(\bar{a})$. It is easy to check that $I' \models def(0,p,\psi)$ and $p^I \subseteq p^{I'}$ since p does not occur in ψ. Finally we have $I' \models \varphi$ since $p^I \subseteq p^{I'}$ and φ is monotone wrt. p. ■

Example 1: Consider a definition

$$def(0,p,\psi) \overset{\text{def}}{=} \forall x\ (p(x) \leftrightarrow (\forall y\ (q(x,y) \leftrightarrow s(x,y)))). \tag{2}$$

Such definitions frequently occur in encodings of hardware verification into first-order logic where, e.g., $p(x)$ can represent equivalence between two bit-vectors $q(x,y)$ and $s(x,y)$ at time x. After Skolemization and clausification of $def(0,p,\psi)$ we obtain clauses outside of the EPR fragment due to non-constant Skolem functions, thanks to the negative occurrence of the $\forall$ quantifier in the positive subdefinition of $def(0,p,\psi)$. Now if all other occurrences of p in our formula are positive we can apply UDE and simplify our definition to

$$def(-1,p,\psi) \overset{\text{def}}{=} \forall x\ (p(x) \rightarrow (\forall y\ (q(x,y) \leftrightarrow s(x,y)))). \tag{3}$$

It is easy to see that after Skolemization of this simplified definition we obtain an EPR formula.

IV. Definition resolution

A *resolvent* of two definitions $def(1,p,\psi)$ and $def(-1,p,\psi')$ is a universal closure of the formula $\psi \rightarrow \psi'$, denoted as $def(1,p,\psi) \otimes def(-1,p,\psi')$. In Table II we define *definition resolution transformation* (DRT) which can be used to eliminate a definition of a predicate based on exhaustive application of resolution. DRT is similar to the variable elimination rule well studied in the propositional case (we refer to [12] for a comprehensive survey of propositional preprocessing techniques).

Theorem 2: DRT is a satisfiability preserving and terminating transformation.

$\varphi \wedge \bigwedge_{i:1 \leq i \leq n} \mathit{def}(1, p, \psi_i) \bigwedge_{j:1 \leq j \leq m} \mathit{def}(-1, p, \gamma_j) \Rightarrow$ $\varphi \wedge \bigwedge_{i,j:1 \leq i \leq n; 1 \leq j \leq m} \mathit{def}(1, p, \psi_i) \otimes \mathit{def}(-1, p, \gamma_j)$, where p does not occur in φ.

TABLE II
DEFINITION RESOLUTION TRANSFORMATION (DRT)

$\varphi[p(\bar{t})]_\pi \wedge \mathit{def}(pol, p, \psi) \Rightarrow \varphi[\psi\sigma]_\pi \wedge \mathit{def}(pol, p, \psi)$, where $\bar{x}\sigma = \bar{t}$, and either (i) $pol = 0$, or (ii) $pol \neq 0$ and all occurrences of p in $\varphi[p(\bar{t})]$ are of polarity $-pol$.

TABLE III
DEFINITION INLINING TRANSFORMATION (DIT)

We can slightly generalise DRT to definitions of the form $\mathit{def}(0, p, \psi)$ by splitting such definitions into positive and negative subdefinitions, applying DRT to the new definitions, and removing tautologies of the form $\psi \vee \neg\psi$.

Let us note that although DRT transformation is terminating, it can quickly increase the size of the formula and therefore is usually applied only in specific cases.

V. Definition inlining

One way of eliminating a predicate definition is to exhaustively *inline* it as defined in Table III. For related discussion we refer to [20] and in the QBF setting to [10].

Theorem 3: DIT is a satisfiability preserving transformation. Moreover, any sequence of DIT applications wrt. a given predicate definition is terminating.

After an exhaustive application of DIT wrt. a predicate definition $\mathit{def}(pol, p, \psi)$ we can eliminate this definition altogether by applying UDE.

Let us note that DIT can quickly increase the size of the resulting formula. We define a special case where such an increase stays linear wrt. size of the formula, called *non-growing definition inlining*.

Definition 1: A predicate definition $\mathit{def}(pol, p, \psi)$ is *non-growing* wrt. a formula φ, if either (i) p occurs only once in φ, or (ii) ψ is an EPR literal. An application of DIT $\varphi[p(\bar{t})]_\pi \wedge \mathit{def}(pol, p, \psi) \Rightarrow \varphi[\psi\sigma]_\pi \wedge \mathit{def}(pol, p, \psi)$ is *non-growing* (NDIT) if $\mathit{def}(pol, p, \psi)$ is non-growing wrt. $\varphi[p(\bar{t})]_\pi$.

Theorem 4: NDIT increases the size of the formula linearly wrt. the number of transformation steps.

Let us note that non-growing inlining is not confluent in general.

VI. EPR restoring inlining

In Section III we saw that UDE can help obtain EPR resulting clausification. It turns out that for many problems, in particular those coming from hardware verification, applying UDE is not sufficient for obtaining EPR resulting clausification. Let us show how DIT can be used to restore EPR resulting clausification.

Example 2: Consider a definition

$$\mathit{def}(0, p, \psi) = \forall x \ (p(x) \leftrightarrow \forall y \ q(x, y))$$

and a formula

$$\begin{aligned}\varphi \quad = \quad & [p(a) \rightarrow (\forall z \ (q(z, c) \leftrightarrow q(d, z)))] \wedge \\ & [\forall u \ (p(u) \vee q(c, d))].\end{aligned}$$

After Skolemization and clausification of $\mathit{def}(0, p, \psi)$, we obtain two clauses $p(x) \vee \neg q(x, sk(x))$ and $\neg p(x) \vee q(x, y)$, corresponding to Skolemization of $\mathit{def}(1, p, \psi)$ and $\mathit{def}(-1, p, \psi)$ respectively. Let us note that the first clause is non-EPR. Moreover, p occurs both positively and negatively in φ and therefore we cannot apply UDE as we did in Example 1.

Let us discuss how one can restore EPR using inlining.

If we inline all non-positive occurrences of p in φ according to $\mathit{def}(0, p, \psi)$, we obtain

$$\begin{aligned}\varphi' \quad = \quad & [(\forall y \ q(a, y)) \rightarrow (\forall z \ (q(z, c) \leftrightarrow q(d, z)))] \wedge \\ & [\forall u \ (p(u) \vee q(c, d))].\end{aligned}$$

Let us note that after inlining, variable x in the definition of p became instantiated by a constant a. Therefore, standard clausification of φ' will result in an EPR formula. Moreover, now all occurrences of p in φ' are positive, and therefore we can apply UDE to $\varphi' \wedge \mathit{def}(0, p, \psi)$, obtaining $\varphi' \wedge \mathit{def}(-1, p, \psi)$. Finally, standard clausification of $\mathit{def}(-1, p, \psi)$ is also in EPR. This example demonstrates how definition inlining in combination with unused definition elimination can be used to obtain an EPR resulting clausification.

Definition 2: A predicate definition $\mathit{def}(pol, p, \varphi)$ is *pre-EPR* if $\mathcal{SK}(\mathit{def}(pol, p, \varphi))$ is not EPR and φ is of the form $Q\bar{y}\psi(\bar{x}, \bar{y})$, where $FV(\varphi) = \{\bar{x}\}$, $Q \in \{\exists, \forall\}$ and ψ is quantifier free.

Let us note that for a pre-EPR predicate definition $\mathit{def}(0, p, \varphi)$, either its positive subdefinition is EPR and its negative subdefinition pre-EPR or vice versa.

A substitution σ is constant-grounding for a set of variables V if σ maps all variables in V to constants.

Lemma 1: Let $\mathit{def}(pol, p, \varphi)$ be a pre-EPR predicate definition. Then $\mathcal{SK}(\varphi\sigma)$ is EPR for any substitution that is constant grounding for $FV(\varphi)$.

The EPR restoring inlining strategy (ERI) consists of exhaustive application of inlining to pre-EPR definitions until pre-EPR (sub)definitions can be eliminated by UDE.

VII. EPR resulting clausification for non-cycling definitions

Consider a set of definitions

$$\mathcal{D} = \{\mathit{def}(pol_1, p_1, \psi_1), \ldots, \mathit{def}(pol_k, p_k, \psi_k)\}.$$

Define a binary dependency relation between symbols as follows: $(p_i, p_j) \in \mathit{dep}$ if and only if p_j occurs in ψ_i. $\mathcal{D}$ is called *non-cycling* if the transitive closure of dep is a strict ordering. $\mathcal{D}$ is called *non-branching* if each predicate has at most one definition in $\mathcal{D}$.

Theorem 5: Consider a formula φ that can be split into $\varphi_{epr} \wedge \mathcal{D}$, where (i) $\mathcal{SK}(\varphi_{epr})$ is an EPR formula, (ii) $\mathcal{D}$ is a set of non-cycling pre-EPR definitions, and (iii) all occurrences of predicates in $\mathcal{SK}(\varphi_{epr})$ are ground-matching in $\mathcal{D}$. Then the EPR restoring inlining strategy is EPR resulting on φ.

In order to obtain an EPR resulting clausification, we need to resort to DIT, which generally does not satisfy our non-growing criterium. In the next sections we consider techniques that simplify definitions and formulas further and are helpful in restoring the non-growing condition in practical cases.

VIII. Argument Collapsing

Consider a first-order formula φ and assume that all occurrences of an m-ary predicate p have distinct constants $c_1, \ldots, c_n$ at the k-th argument (for some k). Moreover assume that

$$\varphi \models c_i \neq c_j \quad \text{for} \quad 1 \leq i < j \leq n. \tag{4}$$

Then we can introduce new $m-1$-ary predicates $p_1, \ldots, p_n$ and replace each occurrence of p where c_i occurs as the k-th argument with the corresponding $m-1$-ary predicate p_i.

This transformation can lead to some equivalences becoming predicate definitions, and therefore eligible for all predicate definition-related transformations. This frequently happens for example in hardware encodings, where some predicate arguments are bit-blasted. Although in general checking condition (4) is as difficult as checking the satisfiability of the formula, in many cases this condition is trivially satisfied. For example, if $c_1, \ldots, c_n$ represent bit-indexes, then when all bit-indexes are enforced to be different this condition is automatically satisfied as in the case of (selective) bit-blasting.

IX. Conditional rewriting

In previous sections of this paper we were addressing unconditional predicate definitions, which were formulas of the form $p(\bar{x}) \star \psi(\bar{x})$, where $\star \in \{\leftrightarrow, \rightarrow, \leftarrow\}$. Some definitions, however, may hold only under certain assumptions; we would call these conditional, and they appear as formulas $\varphi(\bar{x}) \rightarrow (p(\bar{x}) \star \psi(\bar{x}))$. It is not sound to inline such definitions in the entire problem; however, if we have a formula which is also conditioned by φ, or more generally by some φ' such that $\varphi' \rightarrow \varphi$, we can safely perform the inlining there.

Moreover, this observation does not hold only for predicate definitions, but also for equalities. If we have $\varphi(\bar{x}) \rightarrow s = t$, we can use $s = t$ for rewriting terms in formulas conditioned by φ.

As an example, consider a typical formula which occurs in hardware encodings: $next(x,y) \rightarrow (p(x,y) \leftrightarrow \psi(x))$, which informally states that p holds at the consecutive states provided that ψ holds in the current state. Now we can inline this conditional definition of p in other formulas which are also conditioned by the next state predicate. For example $next(x,y) \rightarrow (p(x,y) \wedge (\ p(c,d) \vee q(y)))$ can be rewritten by conditional inlining to $next(x,y) \rightarrow (\psi(x) \wedge (\ \psi(c) \vee q(y)))$.

X. Definition merging

In our experience many problems from hardware formalisations contain predicates which are implicitly equivalent. Such predicates can be merged, considerably speeding up reasoning. More generally, we will consider implied non-growing predicate definitions NDI, shown in Table IV. Let us note that after application of NDI one can exhaustively apply DIT and UDE, eliminating the defined predicate from the problem. NDI covers the special case of implicitly equivalent predicates, since the equivalence of two predicates p and q can be represented as a non-growing definition $def(0, p, q(\bar{x}))$. In the following we consider the case of non-growing definitions of the form $def(pol, p, \psi)$, where ψ is an EPR literal.

$\varphi \Rightarrow \varphi \wedge def(pol, p, \psi)$, where
(i) $\varphi \models def(pol, p, \psi)$,
(ii) $def(pol, p, \psi)$ is non-growing and
(iii) definition inlining is applicable to $def(pol, p, \psi)$ and φ

TABLE IV
Non-growing Definition Introduction (NDI)

In general, checking condition (i) of the applicability of NDI is undecidable, and therefore we need to resort to some heuristics. First we consider syntactic heuristics. Syntactic heuristics will be parameterized by a normalising function. A *normalising function* is a mapping of formulas into equivalent formulas. For example, a function that transforms formulas into negation normal form is a normalising function. There are many other useful normalising functions, e.g., removing double negations or eliminating some connectives. Let us fix a normalising function Δ. Then, two definitions $def(0, p_1, \psi_1)$ and $def(0, p_2, \psi_2)$ are *normalising equivalent* wrt. Δ if $\Delta(\psi_1)$ and $\Delta(\psi_2)$ are syntactically the same formulas.

Let us introduce *syntactic definition merging* as follows. Let $\varphi = \chi \wedge def(0, p_1, \psi_1) \wedge def(0, p_2, \psi_2)$, where ψ_1 and ψ_2 are normalising equivalent. Wlog assume $arity(p_1) \geq arity(p_2)$. Then $\varphi \Rightarrow \varphi \wedge def(0, p_1, p_2(\bar{x}))$ using NDI and we can eliminate p_1 from φ using DIT and UDE.

We implemented the following normalising function Δ_{syn} which (i) transforms formulas into negation normal form, (ii) flattens conjunctions and disjunctions, and (iii) bottom-up renames bound variables, applies sharing of subformulas, and orders conjuncts/disjuncts in disjunctions/conjunctions according to the ordering induced by sharing.

XI. SAT sweeping

In this section we discuss discovery of predicate definitions using propositional reasoning. The problem consists of two tasks. The first task is to convert the first-order problem into a propositional problem so that equivalences found between propositional variables will correspond to equivalences between first-order formulas. The second task is to efficiently find equivalences between variables in a given propositional problem.

Definition 3: Let τ be an injective map of first-order atoms to propositional variables. We extend τ so that it maps unquantified first-order formulas to propositional formulas in a straightforward way (e.g., $\tau(\varphi \wedge \rho) \mapsto \tau(\varphi) \wedge \tau(\rho)$). We further extend τ to universally quantified formulas in prenex form by dropping quantifiers.

Theorem 6: If $\tau(\varphi) \vdash \tau(\rho)$, then it holds that $\varphi \vdash \rho$.

The above theorem is a different formulation of a result given in [15] on under-approximating first-order reasoning using a SAT solver. To apply it to our case, we Skolemize and clausify the problem by using the default $\mathcal{SK}$ and $\mathcal{CL}$ procedures, and use the τ map to translate it to a first-order problem. Any equivalences between propositional variables implied by this problem can then be lifted back to first-order.

Now, given a satisfiable propositional formula φ and a set of interesting propositional variables V, our goal is to discover (some of) the equivalences implied by φ between literals of variables V. For convenience of notation, we consider the propositional constant $\top$ to be one of the interesting variables, which extends the approach also to discovery of true literals. We base our discovery of the propositional equivalences on the idea of simultaneous implicative SAT solving presented in [14], which allows discovery of implied implications (and equivalences) in one call to the SAT solver.

One problem to address is that the clausification process can extend the signature of the formula by introducing new symbols, for example Skolem constants. We use a naive way of dealing with this issue — when an equivalence contains a symbol that is not in the original signature, we discard the equivalence. There may be more advanced ways of eliminating these symbols; however, in our practical applications the presence of introduced symbols did not become a significant problem.

The above approach can be further extended to find equivalences between general sub-formulas, not only between atoms. To this end, we may do an additional transformation on the problem, before it is Skolemized and clausified. First we convert the formula to a QAIG graph (described in Section XII) and then use the Tseitin transformation on the graph, introducing a new name predicate for each node. When we later discover equivalences involving the introduced name predicates, we translate them back into the signature of the input formula by unfolding the introduced names.

XII. QAIG

Following the And-Inverter Graph (AIG) representation [17] of propositional problems widely used in propositional decision procedures, we introduce its counter-part data structure QAIG (Quantified And-Inverter Graphs). It is based on the AIG structure but contains an additional kind of node to represent quantifiers.

The set of QAIG terms on a set of atoms A can be defined as the smallest set of terms Q such that:

$$\begin{array}{l}
\top \in Q \\
\forall a \in A : atom(a) \in Q \\
\forall q \in Q : neg(q) \in Q \\
\forall q_1, q_2 \in Q : conj(q_1, q_2) \in Q \\
\forall q \in Q, x \in \mathrm{free}(q) : quant(x, q) \in Q
\end{array}$$

where $\mathrm{free}(q)$ is the set of free variables in the QAIG q. Below we will use q to denote QAIG nodes.

The QAIG data structure is a canonical in-memory representation of the QAIG terms. Canonicity of the structure means that if two terms are syntactically equal, they are represented by the same memory object. On top of this, we also normalize the order of the arguments in the $conj$ term and eagerly perform the local AIG simplifications proposed in [7].

Lemma 2: An arbitrary first-order formula can be converted to QAIG structure in a single linear-time traversal, assuming a constant-time access to a hash table.

Proof: The conversion is performed by bottom-up application of following transformation rules:

$$\begin{array}{l}
\mathrm{ftq}(a) \Rightarrow atom(a) \text{ for atoms } a \in A \\
\mathrm{ftq}(\phi \wedge \psi) \Rightarrow conj(\mathrm{ftq}(\phi), \mathrm{ftq}(\psi)) \\
\mathrm{ftq}(\phi \vee \psi) \Rightarrow neg(conj(neg(\mathrm{ftq}(\phi)), neg(\mathrm{ftq}(\psi))) \\
\mathrm{ftq}(\exists x : \phi) \Rightarrow neg(quant(x, neg(\mathrm{ftq}(\phi)))) \\
\mathrm{ftq}(\phi \leftrightarrow \psi) \Rightarrow conj(neg(conj(neg(\mathrm{ftq}(\phi)), \mathrm{ftq}(\psi)), \\
\qquad\qquad\qquad neg(conj(\mathrm{ftq}(\phi), neg(\mathrm{ftq}(\psi)))) \\
\ldots
\end{array}$$

Rules for other logical connectives can be written analogously. Each of the rules transforms a formula into QAIG in constant time, assuming that its subformulas are already transformed and that construction of an QAIG term having its arguments is a constant time operation. ■

One thing to note is that in the rule for equivalence we see two occurrences of the $\mathrm{ftq}(\phi)$ (as well as of $\mathrm{ftq}(\psi)$) on the right-hand side. If we were using a flat representation to keep the QAIG terms, applying the rewriting rule would double the size of the term. However, as we use a canonical representation, we are interested in the number of distinct QAIG terms. This number grows only by a constant amount, so the size of the canonical QAIG structure will remain at most linear with the size of the formula.

A. QAIG Inlining

An important goal with the QAIG structure was to obtain a good infrastructure for performing definition inlining without exponential growth in the size of the problem. It can be used for implementing the inlining rules (N)DIT and ERI. QAIGs also provide better sharing of subformulas and definition merging.

At a high level, the QAIG inlining algorithm can be described as follows:

1) Collect the set of candidate rewrite rules $atom(a) \Rightarrow q$: This is done by a scan through the problem, looking for formulas in the shape of $\forall\bar{x}(a(\bar{x}) \leftrightarrow \phi(\bar{x}))$. Here $a(\bar{x})$ denotes an arbitrary non-equality atom with variables $\bar{x}$, and $\phi(\bar{x})$ stands for a formula with free variables being a subset of $\bar{x}$. In order to enable more definitions eligible for inlining, we also apply argument collapsing whenever possible, see Section VIII.
2) Instantiate candidate rules:
Whenever there is a rule in the form $atom(a(\bar{x})) \Rightarrow q(\bar{x})$ and there is an atom $a(\bar{t})$ where $\bar{t}$ is different from $\bar{x}$, we add an instance of the rule $atom(\bar{t}) \Rightarrow q(\bar{t})$ as another candidate rule. Let us note that such instantiated rules are used only for inlining; we do not add them to the resulting QAIG since they are subsumed by the original rules.

3) Remove cyclic dependencies:
   If we have a chain of candidate rules $atom(a_0) \Rightarrow q_0, \dots, atom(a_n) \Rightarrow q_n$ such that $atom(a_n)$ occurs in q_0 and, for each $0 \leq i < n$, $atom(a_i)$ occurs in q_{i+1}, we remove one of the candidate rules to remove the cycle.
4) Apply rules to the QAIG representation of the problem:
   We exhaustively apply generated inlining rules to the QAIG. In this step we must be careful when rewriting using instantiated rules due to variable sharing. In particular, to improve sharing we do not assume that QAIGs are rectified and variable instantiation becomes a non-trivial problem which we consider in the next subsection.

The second step, which involves instantiation, is the only step where the size of the QAIG structure may grow[1] and is potentially the most time consuming. Instantiation is also specific to QAIGs, as the original AIG structure works with propositional atoms where instantiation does not make sense. In the next subsection we focus on the algorithm for QAIG instantiation and discuss some of its properties.

B. QAIG Instantiation

During the instantiation of candidate rules we need to apply a substitution for free variables in a QAIG formula. This cannot be done by a straightforward bottom-up traversal of the QAIG graph, as an atom a may appear at various positions of the QAIG, having different variables bound by its ancestor quantifier nodes. For example take a QAIG

$$conj(atom(p(x)), quant(x, atom(p(x))))$$

In the first occurrence of the atom $p(x)$ the variable is free; however, in the second it is bound by a quantifier. Applying a substitution $\{a/x\}$ will therefore result in

$$conj(atom(p(a)), quant(x, atom(p(x))))$$

We can express the instantiation as a set of rewrite rules parameterized by a substitution:

$$\begin{array}{l} T_\sigma(atom(a)) = atom(a\sigma) \\ T_\sigma(neg(a)) = neg(T_\sigma(a)) \\ T_\sigma(and(a,b)) = and(T_\sigma(a), T_\sigma(b)) \\ T_\sigma(quant(x,a)) = quant(x, T_{\sigma'}(a)), \\ \qquad\qquad \text{where } \sigma' \text{ is } \sigma \text{ with } x \text{ unbound.} \end{array}$$

Lemma 3: If we denote the size of the QAIG structure by n, the size of the QAIG term (which can be exponential with the size of the DAG data structure) by N and the number of variables in the substitution by m, the application of the instantiation rules can be implemented with complexity $O(\min(N, 2^m.n))$.

Proof: The bound $2^m.n$ follows from the fact that with the last rule we may generate at most 2^m possible substitutions, and then we may cache the pairs of a QAIG term and the substitutions applied to it. The bound N is valid because apart from the possible speed up by the earlier mentioned caching, we traverse the QAIG as a term of length N, rather than as a data structure of size n. ■

[1]In the rewriting step we still create new nodes, but for every added node there is a node that was rewritten and therefore removed.

It can be noted that if in no QAIG subgraph would any variable occur as both free and bound, the instantiation could be performed in $O(n)$, as we would know in advance which variables would be instantiated and which would be quantified. Such a representation can be achieved by variable renaming; however, this would decrease the amount of sharing in the QAIG structure. For example, if we consider QAIG

$$conj(atom(p(x)), quant(x, atom(p(x))))$$

its size is 3: the $conj$ node, $quant$ node and the $atom(p(x))$ node which is referred to twice, once by the $conj$ node and once by $quant$. In order to ensure that no variable occurs both as bound and free, we would need to rename one of the occurrences, obtaining

$$conj(atom(p(x)), quant(y, atom(p(y))))$$

Now the size is 4, as we have two $atom$ nodes, $atom(p(x))$ and $atom(p(y))$.

C. QAIG BDD Sweeping

Following the idea of BDD sweeping for propositional problems [17], we attempt to simplify QAIGs using BDDs.

We perform the simplification from simpler AIGs to more complex. When we process an AIG node q, we first check whether it hasn't been simplified into q' by simplifications on its parent nodes. Then, if the number of distinct atoms in the AIG is lower than a given threshold (16 in our implementation), we convert it into a BDD and then back, obtaining q''. If the DAG size of q'' is smaller than the size of q' we use q'' as the simplified node, otherwise we use q'. We also keep a map where for each BDD we store the most compact QAIG representation of it we have encountered. If we encounter several QAIGs with the same BDD, we replace them in the end by the most compact one.

The conversion of QAIGs into BDDs uses a straightforward bottom-up algorithm atb:

$$\begin{array}{l} \mathrm{atb}(atom(a)) = bdd_{var}(atom(a)) \\ \mathrm{atb}(quant(x,q)) = bdd_{var}(quant(x,q)) \\ \mathrm{atb}(neg(q)) = bdd_{neg}(q) \\ \mathrm{atb}(conj(q_1,q_2)) = bdd_{and}(\mathrm{atb}(q_1), \mathrm{atb}(q_2)) \end{array}$$

When converting from BDD to an QAIG, we first extract from the BDD all literals L such that the BDD formula φ can be written as $\varphi \leftrightarrow L \wedge \varphi[L := \top]$ or $\varphi \leftrightarrow L \rightarrow \varphi[L := \top]$. Then we continue with the conversion on the simplified formula $\varphi[L := \top]$. When there are no more possible extractions, we perform a naive conversion

$$\begin{array}{r} \mathrm{bta}(ite(x,t,e)) = and(neg(and(atom(x), neg(\mathrm{bta}(t)))), \\ neg(and(neg(atom(x)), neg(\mathrm{bta}(e))))) \end{array}$$

D. AIG Definition Introduction

We traverse an AIG in a top-down manner, and for each node we remember how many times it would appear in a tree-like representation of the AIG (which can be exponentially large, compared to the DAG representation). If the counter

rule	full name
UDE	Unused Definition Elimination
DRT	Definition Resolution Transformation
(N)DIT	(Non-growing) Definition Inlining Transf.
ERI	EPR Restoring Inlining
NDI	Non-growing Definition Introduction
ED	Equivalence Discovery (or SAT sweeping)
AC	Argument Collapsing
ABS	AIG BDD Sweeping
ADI	AIG Definition Introduction
ACR	AIG Conditional Rewriting
VEP	Variable Equality Propagation

Fig. 1. Simplification transformations

of a node q reaches a certain threshold value (4 in our implementation), we introduce for it a definition $p(\bar{x})$ where $\bar{x}$ are all the free variables of the node q. We will use this definition in place of φ later when we convert the AIG representation back into the non-shared first-order formulas, which will be converted to $p(\bar{x}) \leftrightarrow \varphi_q(\bar{x})$, where $\varphi_q(\bar{x})$ is the formula corresponding to the node q.

E. QAIG Variable Equality Propagation

If a variable occurs in an equality, under some conditions we may propagate it into neighbouring subformulas. We perform this transformation on first-order formulas, but using the above AIG instantiation terminology it can be expressed as

$$and(x = s, b) \Rightarrow and(x = s, T_{x \to s}(b))$$
$$quant(x, neg(and(x = s, b))) \Rightarrow T_{x \to s}(b),$$

where x does not occur in s.

In the first case we cannot remove the equality $x = s$, as x may occur also elsewhere in the problem. However, in the second case (due to the quantifier) we know x does not appear elsewhere, so the equality can be removed. The second rule is also discussed in [23] in the context of simplifying quantified bit-vector formulas.

XIII. Experimental results

Figure 1 summarizes the main simplification transformations discussed in the paper. They are implemented in Vampire's classifier. The implementation is flexible in that these options can be run in a different order, often repeatedly (or until fix-point) when useful.

We have evaluated the simplification techniques reported in the paper on three sets of benchmarks:

(A) EPR-based bounded model checking problems [9].
(B) The QA_UF problems from the SMT library [3].
(C) The FOF problems from the TPTP library [22].

In all the experiments, the time spent on pre-processing was negligible compared to the timeout used and is not reported.

A. Evaluation on EPR-based BMC problems

In [9] we studied an encoding of the BMC [4] problem into first-order logic. The BMC encoding there is called BMC1, as the transition relation is never enrolled explicitly (thus one deals with only one copy of the transition relation). In order to better explain the benchmark results below, let us briefly recall the encoding used for BMC1.

Design	FOF	Bound		CNF size		Time	
block	size	Bln	Dft	Bln	Dft	Bln	Dft
BPB2	913	7	9	1977	2921	6994	8023
DCC1	1093	4	4	3209	1615	5999	8981
DCC2	431	7	10	861	370	6542	9465
DCI1	4678	0	1	15899	9852	149	3085
PMS1	574	5	7	1295	1016	8157	6771
ROB2	713	5	7	1717	1157	8239	6157
SCD1	736	8	9	1908	1328	7704	5366
SCD2	267	8	15	755	524	5691	6370
TOTAL	9404	44	62	27621	18783	49475	54218

TABLE V
BMC1 results on industrial benchmarks.

Let n be a non-negative integer. The *n-step unrolling of the transition system* is defined as follows. Take new constants $s_0, \ldots, s_n$ and a new binary predicate $next$. Denote by $In(\mathtt{S})$, $Fin(\mathtt{S})$, and $Trans(\mathtt{S}, \mathtt{S'})$ the initial and final state constraints, and the transition relation, respectively. The n-step unrolling of the transition system is defined as the set of formulas

$$In(s_0); Fin(s_n);$$
$$\forall \mathtt{S}, \mathtt{S'} (next(\mathtt{S}, \mathtt{S'}) \to Trans(\mathtt{S}, \mathtt{S'}));$$
$$next(s_0, s_1); next(s_1, s_2); \ldots next(s_{n-1}, s_n).$$

In BMC1, it is possible to solve the BMC problems incrementally per bound, and increasing the bound to $n+1$ is expressed by adding an extra constant s_{n+1} and an axiom $next(s_n, s_{n+1})$. (There are a few more subtleties involved in unrolling, but they are irrelevant to the discussion here.)

Table V displays bounds reached by the iProver solver [15] on eight BMC1 benchmarks produced from actual Intel hardware designs. We also report the sizes of the original FOF problems and the sizes of the resulting CNFs, and the solver run-times. The timeout used was 1000 seconds. This data is given for the base-line (or Bln, for short) clausification algorithm of Vampire, and a reasonable default (or Dft for short) version to which we arrived as a result of experiments on BMC1 benchmarks. Unused definition elimination (UDE) is already part of the Vampire baseline clausifier. In the default version above, all the options listed in Figure 1 except for ACR are switched on. (Surprisingly to us, ACR didn't prove useful on BMC1, even if it helps simplifying within formulas ϕ in next-state axioms of the form $\forall \mathtt{S}, \mathtt{S'} (next(\mathtt{S}, \mathtt{S'}) \to \phi(\mathtt{S}, \mathtt{S'}))$.) As can be observed from the table, with the advanced clausification options the total CNF size was reduced from 27621 to 18783, and the number of solved bounds increased from 44 to 62, with only a slight increase in solving time. Note also that higher BMC bounds are much more difficult to solve than lower bounds. Thanks to DIT and ERI, all the resulting CNFs were EPR.

B. Evaluation on SMT benchmarks

We used our clausification algorithms and then passed the clauses in a TPTP format to the Z3 [18] solver which was run

		Bln+ACR	Dft	Dft+ACR	Dft+ACR(ERI)
$\geq 2x$	faster	100	10	74	122
$> 1x$	faster	4890	1527	4847	4941
$\geq 2x$	slower	36	33	36	36

TABLE VI
PERFORMANCE RESULTS FOR QA_UF SMT PROBLEMS.

with a timeout of 30 seconds. Out of 93 problems that timed out with either the baseline or the advanced clausification algorithms, 3 problems were uniquely solved after baseline clausification, and 12 problems could only be solved using the advanced clausification options. Since these represent only a small fraction of the entire problem set, in Table VI we report runtime results, where ACR refers to full conditional rewriting and ACR(ERI) to conditional rewriting restricted to the EPR-restoring strategy. We can conclude that these preprocessing techniques can considerably speed up SMT solvers on a number of problems. We can also note that ACR is very useful both with the baseline and advanced clausification options.

C. Evaluation on TPTP benchmarks

We also evaluated the clausification configurations described in Table VI on all 14540 FOF problems of the TPTP library. We collectively refer to these configurations as advanced clausification configurations. We ran both Vampire and iProver solvers (using Vampire's classifier) with 30 and 60 seconds timeouts, respectively. A decrease in the number of clauses after applying advanced clausification occurred in 2922 problems. All together, with the advanced clausification configurations Vampire solved 276 problems that it cannot solve (with the same strategies) with the baseline clausification, while it solved 76 problems with baseline clausification that cannot be solved with advanced clausification. In total, Vampire solved 11906 problems with advanced clausification configurations while with baseline clausification it solved 11706. Similarly, iProver solved 482 problems only when it used the advanced clausification configurations, and cannot solve 83 problems that it can solve with baseline clausification. In total, iProver solved 7178 problems with baseline clausification and 7577 problems with advanced clausification configurations. We note that 15 (resp. 7) problems uniquely solved with the advanced clausification configurations by Vampire (resp. iProver) have the rating 1 in TPTP 5.3.0; they cannot be solved within a 300 second timeout by any of the solvers that participated in CASC theorem proving competition in 2011.

XIV. CONCLUSIONS

Preprocessing is crucial when dealing with large industrial problems. In this paper we presented a number of preprocessing techniques for simplification of first-order formulas. One of the main goals was to investigate methods for simplifying first-order formulas so that Skolemization and clausification would result in clause sets that are simpler for first-order reasoners. We have investigated methods for definition simplification, EPR-preserving clausification based on definition inlining, discovery and merging of first-order definitions. We also introduced new data structures: quantified AIG, called QAIGs, and a combination of QAIGs and BDDs. We implemented all our techniques in Vampire[2]. Vampire can also be used as an intermediate preprocessing/clausification step for other solvers, in the same way as we used it with iProver and Z3.

We evaluated our techniques over a broad range of benchmarks, including industrial hardware verification benchmarks coming from real-life designs at Intel and largest problem collections for first-order logic (TPTP) and SMT (SMT-LIB). The results are very encouraging, showing that many problems can be solved only with the help of our preprocessing techniques.

There are many directions for future work. Let us only mention that we are planning to develop inprocessing techniques for FOL solvers, that is, we want to combine simplification and reasoning steps more tightly.

REFERENCES

[1] Alberti F., Armando A., Ranise S. ASASP: Automated Symbolic Analysis of Security Policies, CADE 2011.
[2] Baaz M. Egly U., Leitsch A. Normal Form Transformations, in [21], pages 273-333.
[3] Barrett C., Stump A., Tinelli C., The SMT-LIB Standard: Version 2.0
[4] Biere A., Cimatti A., Clarke E., Zhu Y. Symbolic model checking without BDDs, TACAS 1999.
[5] Bjesse P., Boralv A. DAG-aware circuit compression for formal verification, ICCAD 2004.
[6] Brand D. Verification of large synthesized designs, ICCAD 1993.
[7] Brummayer R., Biere A. Local two-level And-Inverter Graph minimization without blowup, MEMICS 2006.
[8] Emmer M., Khasidashvili Z., Korovin K., Voronkov A. Encoding Industrial Hardware Verification Problems into Effectively Propositional Logic FMCAD 2010.
[9] Emmer M., Khasidashvili Z., Korovin K., Sticksel C., Voronkov A. EPR-Based Bounded Model Checking at Word Level, IJCAR 2012.
[10] Giunchiglia E., Marin P., Narizzano M. sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning, SAT 2010.
[11] Hoder K., Kovács L., Voronkov A. Invariant Generation in Vampire, TACAS 2011.
[12] Järvisalo M., Heule M.,3, and Biere A. Inprocessing Rules, IJCAR 2012
[13] Khasidashvili Z., Kinanah M., Voronkov A. Verifying Equivalence of Memories Using a First Order Logic Theorem Prover FMCAD 2009.
[14] Khasidashvili Z., Nadel A. Implicative Simultaneous Satisfiability and Applications, HVC 2011.
[15] Korovin K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR 2008.
[16] Kuehlmann, A. Dynamic Transition Relation Simplification for Bounded Property Checking, ICCAD 2004.
[17] Kuehlmann A., F. Krohm. Equivalence checking using cuts and heaps, DAC 1997.
[18] de Moura L., Bjorner N.: Z3: An Efficient SMT Solver. TACAS 2008.
[19] Navarro-Perez, J.A., Voronkov A. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic, CADE 2007.
[20] Nonnengart A., Weidenbach C. Computing Small Clause Normal Forms, in [21], pages 335-367.
[21] Robinson J. A., Voronkov A. Handbook of Automated Reasoning, Elsevier and MIT Press, 2001.
[22] Sutcliffe G. The 5th IJCAR automated theorem proving system competition @CASC-J5, AI Communications, Volume 24(1), pp. 75-89, 2011.
[23] Wintersteiger C.M., Hamadi Y., de Moura L.M. Efficiently solving quantified bit-vector formulas, FMCAD 2010.

[2]publicly available at http://www.vprover.org/

A Liveness Checking Algorithm that Counts

Koen Claessen
Chalmers University of Technology
koen@chalmers.se

Niklas Sörensson[†]
Mentor Graphics Corporation
niklas_sorensson@mentor.com

***Abstract*—We present a simple but novel algorithm for checking liveness properties of finite-state systems, called k-LIVENESS, which is based on counting and bounding the number of times a fairness constraint can become true. Our implementation of the algorithm is completely SAT-based, works fairly well in practice, and is competitive in performance with alternative methods. In addition, we present a pre-processing technique which can automatically derive extra fairness constraints for any given liveness problem. These constraints can be used to potentially boost the performace of any liveness algorithm. The experimental results show that the extra constraints are particularly beneficial in combination with our k-LIVENESS algorithm.**

I. INTRODUCTION

LTL properties for model checking are traditionally partitioned into two sets: *safety* and *liveness* properties. Roughly, safety properties are properties for which all possible counter examples are finite traces. Liveness properties can have infinite counter examples that are impossible to make finite.

Safety properties are more commonly used in practice, easier to understand, and theoretically easier to check than liveness properties. However, liveness properties still play an important role on many verification projects. The SAT Revolution in model checking at the end of the 1990s [8] gave us new ways of battling the "blow-up" problems associated with typical BDD model checking algorithms, and sparked off a long sequence of new SAT-based safety checkers [11], [4], [10], [5]. On the liveness side, no such explosion of new algorithms took place. (Interesting to mention in this context is that the original paper on Bounded Model Checking [2] treats safety properties and liveness properties equally.)

Related Work The first practical complete SAT-based liveness checking approach was made possible by the *liveness to safety* (LTS) translation by Biere et al. [1]. LTS translates any liveness checking problem into a safety checking problem, after which any safety checker can be used, including SAT-based checkers. The second complete method for liveness, by Bradley et al., called FAIR [6], was only published last year. It consists of a dedicated liveness algorithm, based on a symbolic exploration of the state space using a SAT-solver, looking for strongly connected components. FAIR performed rather well in the Hardware Model Checking Competition 2011 [3], proving more liveness properties than any other participant in the competition.

This paper presents a new model checking algorithm for full LTL, called k-LIVENESS, that is amenable for a SAT-based implementation. Like LTS, k-LIVENESS translates liveness checking into safety checking, but it generates an (infinite) sequence of safety problems rather than just one safety problem. If one of the safety problems in the sequence can be shown to hold, then the original liveness property holds as well. In principle, any safety checker can be used to solve the problems in the sequence of problems, but we implemented and use a SAT-based incremental safety checker especially for this purpose.

As we shall see, k-LIVENESS is a surprisingly simple algorithm, much simpler than FAIR, and arguably also simpler than LTS, but it performs nonetheless quite well when compared to these other algorithms. A drawback of our chosen approach is that, although it is complete for proving as well as disproving LTL properties, it is not suitable in practice yet for finding counter examples. Therefore, there is a need to combine it with a dedicated counter example finder, for example one based on Bounded Model Checking [2].

The second contribution of the paper is a preprocessing step that automatically adds extra fairness constraints to a given liveness problem. The addition of these constraints is sound; the validity of properties of the original circuit is not changed by these extra constraints. The potential benefit is that, depending on the liveness checking algorithm used, these extra fairness constraints make many liveness problems much easier. Our experimental results show that k-LIVENESS in particular, but also LTS benefits very much from these extra constraints.

II. PRELIMINARIES

A trace t is a function from time point and signal name to Boolean value:

$$t : \mathbf{N} \times \text{Signal} \to \mathbf{B}$$

We will use LTL formulas containing operators $\Box, \Diamond, \mathsf{next}\ , \vee, \wedge, \to, \neg, =$ having their standard meaning. We write $S \vdash \phi$ when the system S satisfies the property ϕ, and we write $S, \psi \vdash \phi$ when the system S makes ϕ true under the assumption ψ.

When performing model checking, we assume that the LTL property ϕ at hand has already been translated into a *liveness signal* q, such that

$$S \vdash \phi \qquad \Leftrightarrow \qquad S \vdash \Diamond\Box q \tag{1}$$

In other words, in order to deal with full LTL, we only need to consider proving LTL properties of the form $\Diamond\Box q$. (This is

[†]The bulk of this work was carried out at Chalmers University.

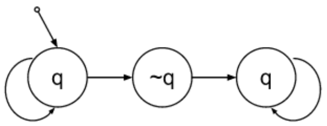


Fig. 1. Counter-example showing that the simple algorithm does not work.

standard [12]; here, q is the negation of the acceptance signal of the Buchi automaton for $\neg\phi$. The above right-hand side is sometimes expressed using the *fairness signal* $\neg q$ as follows:

$$S, \Box\Diamond\neg q \vdash \textbf{false}$$

but we prefer the expression used in (1) above.)

In some contexts, a property ϕ may be translated into several liveness signals, in which case we need to prove their disjunction:

$$S \vdash \phi \qquad \Leftrightarrow \qquad S \vdash \bigvee_i \Diamond\Box q_i \tag{2}$$

In this case, we first combine all liveness signals q_i into one liveness signal q, and then proceed with the liveness signal q. This combination can be done in many (standard) ways; the simplest one is to introduce one auxiliary register for each liveness signal q_i that keeps track of whether that signal has been 0 yet. If all q_i have been 0 at least once, q also becomes 0 and we reset all auxiliary registers. This construction introduces n extra registers and $O(n)$ extra gates for combining n liveness signals.

III. K-Liveness

In this section, we present our basic algorithm for checking liveness, called k-Liveness.

A. *A simple algorithm for $\Diamond\Box q$ (that does not work)*

Consider proving the eventuality property

$$S \vdash \Diamond q$$

for a boolean signal q. For finite state systems, we may prove this by searching for a natural number k such that

$$S \vdash \bigvee_{i \in 0\ldots k} \mathsf{next}^i\ q$$

Indeed, if $\Diamond q$ holds, we can always find such a k. The gain is that, for a given k, the above proof obligation is a safety property, which can be checked by a safety checker. A simple checking algorithm thus tries $k = 0, 1, 2 \ldots$ until $\bigvee_{i \in 0\ldots k} \mathsf{next}^i\ q$ holds.

One might be tempted to try a similar idea for checking the more general safety property

$$S \vdash \Diamond\Box q$$

Alas, there are finite state systems for which the above holds, but for which there is no k such that

$$S \vdash \mathsf{next}^k\ \Box q$$

An example of such a system and property is shown in Fig. 1. Clearly, $\Diamond\Box q$ holds for all traces accepted by the system, but for every k there is a trace such that q becomes false after k steps. So, this method is sound but not complete.

B. *A simple, correct algorithm for checking $\Diamond\Box q$*

Instead of counting (and bounding) the number of clock cycles until the signal q must become true forever, we can instead count (and bound) the number of times k the signal q can be false. If we can find such a bound k, then q has to eventually become true forever. Moreover, if q is a valid liveness signal for a finite state system, we can always find such a k. In the system in Fig. 1, the bound is 1; q can only become false once in every trace.

What we want is expressed more formally by the following lemma.

Lemma 1: Given a finite-state system S for which we have $S \vdash \Diamond\Box q$. Then, there exists a k such that for any trace t of S, there are at most k different points in time i for which $t(i, q) = 0$.

Proof. Assume the opposite: for any k there is a trace t where q becomes false at least k times. Now, pick any k larger than the number of states in S; there must be a trace t in which q becomes false at least k times. Consequently, there must be two different time points i and j for which $t(i) = t(j)$ and $t(i, q) = t(j, q) = 0$, since not all states where q is false can be unique. We can now construct a looping trace t' (obtained from t by repeating the states between i and j) for which q is false infinitely often, which contradicts our original assumption that $S \vdash \Diamond\Box q$. $\Box$

The experimental observation we make is that, in practice, k is often very small (see Fig. 9 in Sect. V), which suggests that finding k might be a practical method for checking liveness properties.

Our algorithm works as follows. We start by setting $k := 0$. Now, we try to show that q can only become false at most k times (this is a safety property). If we succeed, we are done; the property holds. If we fail, we increase k by 1 and try again.

Because of the above lemma, this algorithm is complete for valid properties. However, it does not terminate for properties that are not valid. Theoretically, if we keep finding counter examples for growing k, at some point there must be a trace which contains a repeated state at the appropriate place, thus forming a valid counter example for the original liveness signal. However, this is unlikely to work well in practice. In order to get a complete algorithm also for false properties in practice, a dedicated counter example finding method is used in parallel or in lock-step with k-Liveness.

C. *Implementation*

In our implementation, instead of repeatedly calling a safety checker every time we change k, we use an incremental safety checker. The incremental safety checker proves or disproves a given safety property, after which we can add some more logic and registers to the circuit, and continue with a new safety property. The incremental checker keeps its internal state

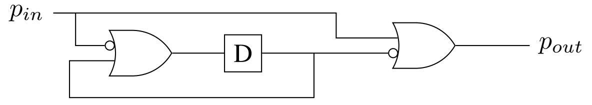


Fig. 2. Absorbing one 0 from a liveness signal (initial state for D is 0)

between calls, so it can reuse information about reachabilty it discovered in earlier runs in later runs too.

Making an existing safety checker incremental is more or less difficult, depending on the underlying algorithm that the checker is based on. The safety checker we used was our own implementation of Bradley's SAT-based safety checking algorithm as implemented in IC3 [5]. Bradley's safety algorithm maintains a finite sequence of sets of clauses, each set belonging to a concrete point in time. It turns out that the algorithmic invariants between these sets are completely independent of the property one is currently proving. So, if the current property has been proven or disproven, we can keep all sets of clauses for the next run of the checker, even though we might add some new logic and change to a new property. The algorithm also maintains a "current depth" counter which does not need to be reset when a new property is checked. Thus, Bradley's safety algorithm is a very nice fit for the liveness checker we are building.

The liveness algorithm starts with $k = 0$, and so the safety property p we have to consider is actually the liveness signal q. So, we start by trying to prove that q can never become false. If this is disproved, we want to increase k and run again. We implement increasing k by 1 by attaching the *absorbing* circuitry shown in Fig. 2 to the safety property p we have just disproved. If p is fed as its input p_{in}, a new safety signal p_{out} is created that behaves just like the previous signal p_{in}, except that it absorbs the first 0 that is produced by its input and turns it into a 1. So, adding the absorbing circuit in the figure and checking its output as the new safety property has the same effect as increasing k by 1. If we disprove the new safety property p_{out}, we attach yet another copy of the circuit to it, and so on, until we have attached enough copies of the absorbing circuit to smother all possible 0's (or we go on forever).

Making Bradley's algorithm incremental amounted to adding only about 30 lines of C++ code to our original implementation[1]. The effect of using an incremental checker is evaluated in Sect. V.

What is presented in this section leaves us with a basic liveness checking algorithm that performs reasonably well (see Sect. V for more details), but there are some bottlenecks, especially when k needs to be large. The next section presents a pre-processing step that greatly boosts the performance of the basic algorithm, and has the potential of improving other liveness checking algorithms as well.

[1]In this way, we ended up with a model checker that can check multiple safety and liveness properties simultaneously, something we have not seen before. We have however not experimentally evaluated the advantages and disadvantages of actually using the model checker as such.

IV. Automatic constraint extraction

Suppose we are checking the following proof obligation:

$$S \vdash \phi \tag{3}$$

Here, ϕ can be a safety property as well as a liveness property. Automatic constraint extraction may construct a new formula ψ which may be used as a constraint (an assumption), thus:

$$S, \psi \vdash \phi \tag{4}$$

The constraint extraction is correct if and only if the proof obligations 3 and 4 are equivalent. The hope is that a model checker may benefit from making use of the constraints ψ.

For safety checking, this idea has been proposed before, e.g. in [7]. As far as we know, we are the first to explore this in the context of liveness checking.

The ideas described here are very much inspired by the algorithm that finds so-called *arenas* in [6]. The idea behind arenas is to divide the state space of the system up into partitions, such that any trace of the system will eventually end up and stay in one such partition only. Arenas are an intricate part of the liveness checking algorithm in [6]; we decoupled the idea from the algorithm, generalized and improved the idea somewhat, and repackaged it as a pre-processing technique for liveness algorithms in general.

A. Stabilizing constraints

The kind of constraints we are going to extract are of the form

$$\Diamond\Box s$$

such that

$$S \vdash \Diamond\Box q \quad \text{iff.} \quad S, \Diamond\Box s \vdash \Diamond\Box q$$

We call constraints of the form $\Diamond\Box s$ a *stabilizing constraint*.

As observed in [6], it turns out that many liveness problems, for example liveness problems involving counters, admit such stabilizing constraints. The first observation we make is directly inspired by [6]: If we find a signal x that is *monotonic*, in other words such that:

$$S \vdash \Box(x \to \mathsf{next}\ x)$$

then it is safe to add $\Diamond\Box(x = \mathsf{next}\ x)$ as a stabilizing constraint. The reason is that for any trace t of S, x must either be 0 all the time, or become 1 at some point and then stay 1 forever. In both cases, we have that eventually, x will keep its value.

Next, we generalize this observation for signals x that are *eventually monotonic*.

Lemma 2: Given a system S and a signal x. If we have

$$S \vdash \Diamond\Box(x \to \mathsf{next}\ x)$$

then we may use $\Diamond\Box(x = \mathsf{next}\ x)$ as a (stabilizing) constraint.

B. Making use of the property

We can also make use of the original liveness signal q we want to prove. Assume that we are solving the liveness problem:

$$S \vdash \Diamond\Box q$$

and that we have found a stabilizing constraint $\Diamond\Box(x \rightarrow \textsf{next}\ x)$. However, suppose we also find out that:

$$S \vdash \Diamond\Box(x \rightarrow q) \tag{5}$$

then it is safe to assume, not only that x will eventually stabilize to some value (0 or 1), but also the much stronger, that x will stabilize to 0! In other words, we can add $\Diamond\Box\neg x$ as a stabilizing constraint and check:

$$S, \Diamond\Box\neg x \vdash \Diamond\Box q$$

instead. Why? Because there are two cases: either x stabilizes to 0 or to 1. If x stabilizes to 1, we already know (because of (5)) that q stabilizes to 1 also, and we have shown the original property. The only interesting case left is when x stabilizes to 0, which is the one we add.

Similarly, if we find out instead of (5) that

$$S \vdash \Diamond\Box(\neg x \rightarrow q)$$

we can add $\Diamond\Box x$ as a stabilizing constraint.

C. Multiple stabilizing constraints

Stabilizing constraints may be used to help find other stabilizing constraints. So, if we have found the stabilizing constraint $\Diamond\Box(x = \textsf{next}\ x)$ by showing:

$$S \vdash \Diamond\Box(x \rightarrow \textsf{next}\ x)$$

then we may use it when considering another candidate y:

$$S, \Diamond\Box(x = \textsf{next}\ x) \vdash \Diamond\Box(y \rightarrow \textsf{next}\ y)$$

In general, we may have found a *set* of stabilizing constraints that we can use to derive new stabilizing constraints, which in turn can give rise to even more stabilizing constraints.

D. Approximating stability checking

In general, when looking for stabilizing constraints as described above, we are asking questions of the following shape:

$$S, \bigwedge_i \Diamond\Box a_i \vdash \Diamond\Box b$$

Here, the a_i are the stabilizing constraints we have already found, and b is a proof obligation that may give rise to a new stabilizing constraint. In order to *decide* questions like the above, we would need a liveness checker, which would defeat the purpose of using this as a pre-processing step to a liveness checker!

Instead, we *approximate* the answer to this question by using a SAT-solver which only talks about two consecutive states of S. We assume we are at a state in the trace where all stability constraints a_i have already become true, and then ask if the desired stability constraint b is now also true. Two states is enough since every a_i and also b contains at most one $\textsf{next}$ operator. We add the assumptions a_i to the first state, and then ask the SAT-solver if b also holds. If the answer from the SAT solver is yes, we know the constraint holds. It is a crude approximation, but very fast and quite effective in practice.

E. Algorithm

In the overall constraint extraction algorithm, we work with a set of circuit points P and a set of found stabilizing constraints M. Initially, the set of found stabilizing constraints M is empty, and the set P consists of all internal points in the circuit (and their negations).

The derivation algorithm works as follows.

For all points x from the set P, we try to prove (using our overapproximation):

$$S, M \vdash \Diamond\Box(x \rightarrow \textsf{next}\ x)$$

If this succeeds, we add the stabilizing constraint $\Diamond\Box(x = \textsf{next}\ x)$ to M, and remove x from P. We then also try to prove (also using the overapproximation):

$$S, M \vdash \Diamond\Box(x \rightarrow q)$$

If this succeeds, we add $\Diamond\Box\neg x$ to M. If not, we try:

$$S, M \vdash \Diamond\Box(\neg x \rightarrow q)$$

If this succeeds, we add $\Diamond\Box x$ to M.

If we discover any new stability constraints, we go through all points x in P again in a new round. If no new stability constraints have been found, we terminate with M.

Note that we actually have a choice of what set of points P we start with. In our experimental results (see Sect. V), we have compared starting with all internal points (and their negations) of the circuit, as well as just having the registers (and their negations), which may be cheaper[2].

F. Making use of stabilizing constraints

Once we find the set M, we can add each of these as constraints, if the model checker can handle such constraints. However, our model checker only handles a single liveness signal q, so here we describe how we can deal with this.

The first step is to turn constraints in M of the form $\Diamond\Box(x = \textsf{next}\ x)$ into a stabilizing constraint with just a Boolean signal c: $\Diamond\Box c$. This is cheap and easy if x is the output of a register (or its negation), because then points representing x and $\textsf{next}\ x$ already exist, and we just create one extra XOR-gate. But if x is an internal point, we may have to introduce extra logic or perhaps even a register to represent c. We might not be willing to pay this price, in which case we can just throw away the constraint $\Diamond\Box(x = \textsf{next}\ x)$.

So, here we have another choice of parameter to the algorithm: Do we keep constraints of the form $\Diamond\Box(x = \textsf{next}\ x)$ even if x is not a register? We have also compared this choice in our experimental results. Note that it may actually

[2]Bradley et al. restrict themselves to registers in their arena discovery method [6].

be beneficial to start with P being all internal points, even if in the end we keep only the constraints on registers. This is because finding constraints on internal points may help in finding more constraints on registers.

Once all constraints in M are of the form $\Diamond\Box c_i$ for Boolean signals c_i, we can turn these into one big constraint $\Diamond\Box(\bigwedge_i c_i)$. This is because $\Diamond\Box$ distributes over $\wedge$. So, we are now checking:

$$S, \Diamond\Box(\bigwedge_i c_i) \vdash \Diamond\Box q$$

which is equivalent to

$$S, \Diamond\Box(\bigwedge_i c_i) \vdash \Diamond\Box((\bigwedge_i c_i) \to q)$$

for which it is enough to check

$$S \vdash \Diamond\Box((\bigwedge_i c_i) \to q)$$

since $\Diamond\Box(\bigwedge_i c_i)$ is a correct extracted constraint.

As we can see, $\bigwedge_i c_i \to q$ is a much weaker liveness signal than q, and therefore it can become false a lot less often, reducing (in many cases significantly) the value of k needed for k-LIVENESS.

V. EXPERIMENTAL RESULTS

In this section, we present an experimental evalation of implementations of the three algorithms discussed in this paper: k-LIVENESS, LTS, and FAIR. The implementations of k-LIVENESS and LTS were made by ourselves, and are based on the same safety checker. The implementation of FAIR we used was made by the original authors. Our implementation of LTS seems to be slightly better than the one used in [6], which explains the differences in evaluations in this paper and [6].

We have run a number of variants of these algorithms on a public set of liveness benchmarks, obtained from the Hardware Model Checking Competition [3]. From that set, we discarded a few problems that were not solvable by any algorithm; a total of 52 problems were solvable by at least one of the tested checkers.

All experiments were run on a cluster of quad-core Intel Xeon E5620 CPUs clocked at 2.4 GHz. The detailed results of most experiments reported here are presented later in the table in Fig. 9. In the table, a dash (—) represents a time-out. The table also contains the values of the k's that were needed for the various versions of k-LIVENESS, including the k's that were reached in case of a time-out.

To get a baseline, we start our evaluation by comparing the *basic* versions of each of the three algorithms, where no fairness constraint extraction is performed. To this end, we made a change to the source code of FAIR, and switched off the arena finding part of their algorithm, which roughly corresponds to our fairness constraint extraction. This arguably crippled (!) version of FAIR is called fair-snd0 in the tables.

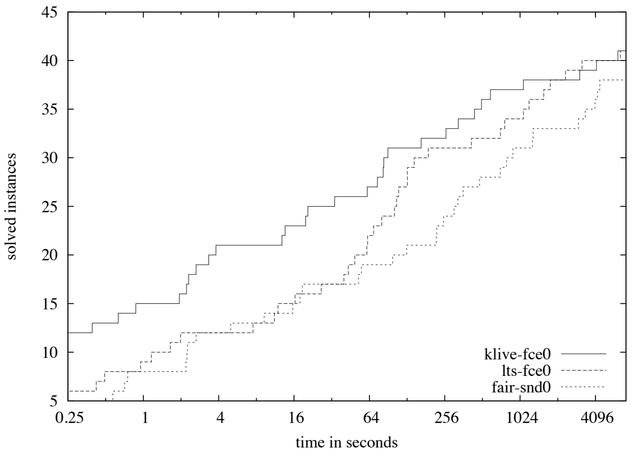


Fig. 3. Comparison of core algorithms

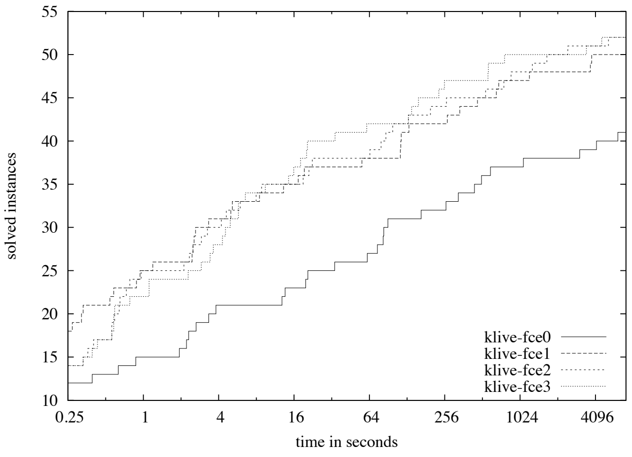


Fig. 4. Effect of stabilizing constraints on k-LIVENESS

Fig. 3 presents this comparison in the form of a cactus plot[3], where we display time outs vs. number of solved problems. We can see here that all three algorithms perform comparably (k-LIVENESS and LTS solve the same amount of problems for the maximum time out). k-LIVENESS performs slightly better than the others at time-outs around a minute or so.

The second comparison we make is about the effect of the various versions of the fairness constraint extraction on the algorithms k-LIVENESS and LTS. We chose not to include FAIR in this comparison, because the original uncrippled version of FAIR already has a similar technique built-in. The results are displayed as cactus plots in Fig. 4 and Fig. 5. Here, fce0 means no constraint extraction, fce1 means constraint extraction only for registers, fce2 means constraint extraction for all points in the circuit, but only added for registers, and fce3 means full constraint extraction and addition for all

[3]Our cactus plots might be considered slightly non-standard; the time axis is at the bottom (where time axes should be) and is logarithmic (since when comparing running times it is the factor, not the difference, which is important).

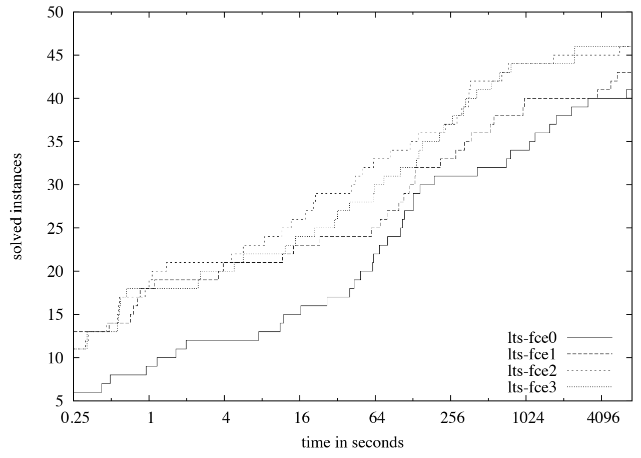


Fig. 5. Effect of stabilizing constraints on LTS

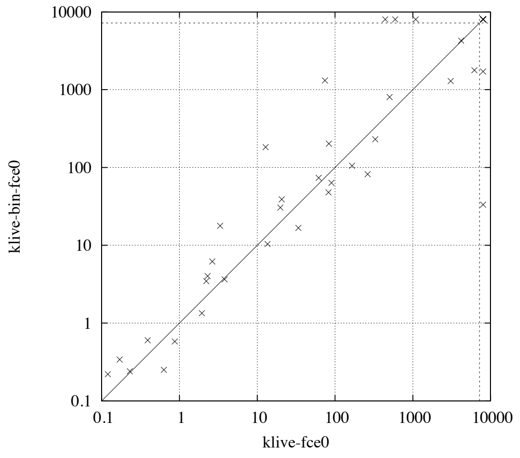


Fig. 7. klive using a binary counter

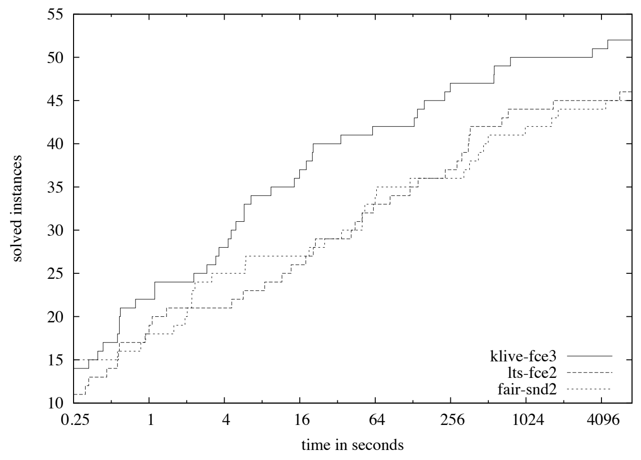


Fig. 6. Comparison of algorithms with best stabilizing constraints

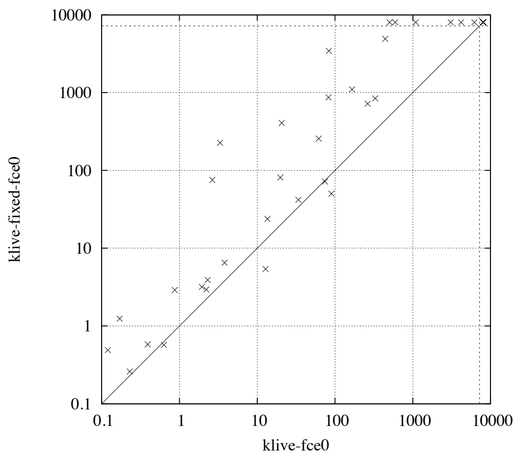


Fig. 8. Non-incremental klive with perfect guessing

circuit points. The conclusion we draw from these graphs is that fce2 and fce3 work best, but it is hard to decide which of those is best based on the set of benchmarks we have.

From Table 9 we can see that when the analysis has a big improvement on the total running time, it is mainly due to a significant reduction in k.

The times reported here is the total time of first running the analysis and then the model checking algorithm. We have not reported detailed results on the running times of the analysis alone. However, in the vast majority of the benchmarks the running time of the analysis is negligible (less than 1 second). Only in a couple of benchmarks did the analysis take a significant amount of time relative to the total time. This indicates that it is possible to find classes of circuits where the analysis as implemented today will not scale up.

The third comparison we want to make is between the best versions of the three algorithms. For k-LIVENESS and LTS, we (rather arbitrarily) chose fce3 and fce2, respectively. For FAIR, we run the original unmodified algorithm. The cactus plots for this comparison are displayed in Fig. 6. We can see that with the fairness constraint extraction on, k-LIVENESS outperforms the other two, who in turn perform remarkably similar.

Finally, we would like to experimentally answer two questions that naturally arise in connection with the k-LIVENESS algorithm. The first question is: For large k, it seems problematic to use an approach that adds circuitry linear in k. What happens when we use a binary counter instead? It turned out it was quite simple to change our algorithm to double k at every incremental step by using a binary counter. The comparison with the original, linearly growing algorithm is displayed in Fig. 7. We can see that some problems indeed are solved a bit faster, but many more problems are solved quite a bit slower. That difference is clearest for fce0, which is why we chose to show that version in the figure. The conclusion is that it might be beneficial to use a binary counter for problems that need a large k, but it is a bad idea to pick this as the default method.

The second question is: Suppose we had a way to (almost) perfectly guess the right k on beforehand. Could we base

name	klive-fce0		klive-fce2		klive-fce3		lts-fce0	lts-fce2	lts-fce3	fair-snd0	fair-snd2
	time	k	time	k	time	k	time	time	time	time	time
arbi0s08p03	442	9	1676	7	567	7	—	359	324	—	—
cuabq2f	0	3	0	2	0	2	62	355	75	5	3
cuabq2mf	0	3	0	2	0	2	2	5	5	1	0
cuabq4f	2	5	2	5	4	5	6474	5739	—	485	121
cuabq4mf	0	5	1	5	1	5	189	44	223	2	2
cuabq8f	74	9	98	9	158	9	—	—	—	1285	—
cuabq8mf	13	9	19	9	16	9	—	—	—	219	66
cucnt10	—	625	0	0	0	0	40	0	0	710	0
cucnt128	—	622	6	0	6	0	—	6	6	—	1
cucnt12	—	614	0	0	0	0	713	0	0	3393	0
cucnt32	—	533	0	0	0	0	—	0	0	—	0
cucnt3	0	0	0	0	0	0	0	0	0	0	0
cufq1	3069	9	2464	8	34	4	—	—	2506	—	—
cugbak	83	32	87	32	131	32	1207	738	2504	127	50
cugcd	3	16	1	5	2	5	1	1	12	2	2
cujc128	—	1960	21	0	20	0	—	21	21	—	2
cujc12	—	1928	0	0	0	0	61	0	0	52	0
cujc32	—	2061	1	0	1	0	—	1	1	—	0
culock	21	83	4	31	7	31	11	12	40	9	6
cunim1	593	60	0	0	0	0	43	0	0	18	0
cunim2	1088	60	3	0	6	0	418	18	265	—	1011
cuom1	—	386	1	0	1	0	—	1	1	—	—
cuom2	—	517	8	0	20	0	1785	50	32	—	64
cuom3	—	866	1	0	1	0	1090	1	1	—	2
cusarb16	0	15	0	0	0	0	0	0	0	2	0
cusarb32	3	31	0	0	0	0	8	0	0	19	0
cutarb16	503	159	131	79	139	76	146	368	209	221	25
cutarb32	—	261	5207	191	4599	188	—	—	—	—	362
cutarb4	1	23	1	11	1	8	1	1	3	1	0
cutarb8	20	63	9	31	9	28	12	14	15	16	2
cutf1	166	8	197	8	14	4	1583	315	777	898	327
cutf3	0	3	1	0	1	0	26	1	2	3	1
cutq1	4176	9	866	7	18	3	2360	—	63	4061	4433
lmcs06abp4p1	2	3	2	3	4	3	3199	20	—	326	34
lmcs06abp4p2	2	4	3	4	3	4	771	1691	626	304	511
lmcs06abp4p4	0	1	1	1	1	1	2	8	30	251	6
lmcs06bc57sp1	14	4	22	3	3	2	80	41	414	—	53
lmcs06bc57sp2	34	2	2	0	5	0	110	232	539	98	50
lmcs06bc57sp3	4	1	5	0	5	0	16	656	137	55	19
lmcs06brp0	0	1	0	0	0	0	0	0	0	—	2
lmcs06brp2	1	2	0	0	0	0	106	1	1	798	—
lmcs06counter0	0	7	0	6	0	5	0	0	0	0	0
lmcs06dme3p2	6193	2	1283	2	3467	2	—	—	—	—	—
lmcs06mutex0	0	2	0	0	0	0	0	0	0	0	0
lmcs06prodcell2	90	19	79	18	61	17	49	62	338	358	469
lmcs06prodcell3	82	80	65	25	230	25	69	288	102	4260	1849
lmcs06prodcell4	62	60	264	24	255	24	102	84	62	1300	426
lmcs06prodcell5	328	109	760	106	769	105	129	121	152	4442	—
lmcs06prodcell6	262	109	543	106	571	105	129	141	143	2999	1643
lmcs06ring0	0	4	0	4	0	4	0	0	0	0	0
lmcs06short0	0	1	0	0	0	0	0	0	0	0	0
lmcs06srg5p0	0	6	0	5	0	5	0	0	1	1	1

Fig. 9. Detailed experimental results.

an algorithm on that? We ran an experiment where we first computed the right k for each benchmark, and then ran the algorithm again, jumping to that k immediately. The results are displayed as a scatterplot in Fig. 8. We can see here that solving the problem for the correct, fixed k directly is much slower than the incremental approach. This is surprising because the incremental approach actually proves more; it also proves that none of the other k are large enough. One explanation is that the state space exploration we force the algorithm to go through when considering lesser k actually helps in finding the proof for the right k. This might also partially explain why the binary counter approach is worse than the linear approach, because it jumps over many k at once as well. However, these explanations are merely speculations and more investigation is needed to fully understand these results.

VI. Discussion and Conclusions

We have presented a new, simple liveness algorithm, called k-LIVENESS, based on finding a limit k on the number of times a fairness signal can become true, that compares favorably against existing liveness algorithms. Finding the right limit k is done by using an incremental safety checker. Our experiments show that the incrementality of the approach is crucial for its efficiency.

Moreover, we developed a preprocessing technique that is heavily inspired by the FAIR algorithm, that can boost the performance of liveness algorithms in general. The main differences between the pre-processing presented here and the arena analysis as part of the FAIR algorithm are: (1) our approach works on all points in the circuit rather than just the registers, (2) our approach makes use of the liveness signal in a different (stronger) way, (3) our approach generates extra constraints separately from the main model checking algorithm.

We evaluated the preprocessing technique positively in particular for k-LIVENESS and LTS. Our experiments show that it is important for the preprocessor to take all internal points of the circuit into account, not only the registers. k-LIVENESS plus fairness constraint extraction performs best in our experiments. It seems that the Achilles heel of k-LIVENESS, namely when the needed k is growing too large, is nicely covered by the fairness constraint extraction, which works very well for counter-like sub-circuits.

The resulting algorithm is arguably much simpler than FAIR, even when taking the preprocessing analysis step into account. Moreover, FAIR is non-deterministic by design (and by necessity), which our algorithm is not.

A drawback of our approach is that it does not seem practical to extract counter examples. To make a model checker that is complete in practice even for false properties, the method needs to be combined with a dedicated counter example finder, for example based on Bounded Model Checking [2]. Our tool Tip, which implemented k-LIVENESS (without preprocessing) in lock-step with a simple BMC method actually won the overall liveness track of the Hardware Model Checking Competition in 2011 [3], showing that this is not a problem in practice. It is future work to investigate how to practically extract possible counter examples from failed safety checks.

For more future work, we intend to investigate the effect that alternative choices have on the efficiency of the algorithm. For example, the circuit in Fig. 2 that is added at each incremental step can be implemented in many different ways, for example by using a shift register. We do not know how changing this circuit affects the performance.

Moreover, we want to find out what effect our preprocessor has on other liveness algorithms, for example BDD-based algorithms.

A more open question is wether it is possible to make an efficient SAT-based model checker for CTL. Ideas from [9] seem promising in this regard.

ACKNOWLEDGMENTS

We would like to thank Johan Mårtensson and Nir Piterman for initial discussions and insights on this work. The anonymous referees' comments were also highly appreciated. Finally, many thanks to the Colorado team (Aaron Bradley, Fabio Somenzi, and Zyad Hassan) for helping us to run their code.

REFERENCES

[1] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *Proc. of Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of Conference on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[3] Armin Biere and Keijo Heljanko. Hardware model checking competition, 2011. Associated with FMCAD'11. http://fmv.jku.at/hwmcc11/.

[4] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[5] Aaron Bradley. SAT-based model checking without unrolling. In *Proc. of Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Lecture Notes in Computer Science. Springer Verlag, 2011.

[6] Aaron Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2011.

[7] 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 Conference on Design Automation and Test in Europe (DATE)*, 2009.

[8] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. Sat-solving in practice. In *Proc. of Workshop on Discrete Event Systems (WODES)*. IEEE, May 2008.

[9] Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In *Proc. of International Conference on Computer-Aided Verification (CAV)*, 2011.

[10] Ken McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. of Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[11] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[12] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. of Symposium on Foundations of Computer Sciece (FOCS)*, 1983.

A Formal Model of a Large Memory that Supports Efficient Execution

Warren A. Hunt, Jr. and Matt Kaufmann
Dept. of Computer Science, University of Texas, Austin, TX 78701
Email: {hunt,kaufmann}@cs.utexas.edu

Abstract—**The validation and application of formal processor models benefits fundamentally from both efficient execution and automated reasoning about the models. We present a memory model written in the ACL2 logic, with both reasoning support and a runtime environment, that accomplishes these objectives. Our memory model provides a space-efficient implementation for an address space of 2^{48} bytes, and is used in our development of an ISA model for x86 instructions. We define and prove invariants, and we use them to prove useful lemmas and to formally verify absence of run-time simulator errors. Our memory model also supports efficient execution through constant-time read and write access in an applicative setting.**

I. Introduction

We describe a model of memory suitable for specifying and simulating a 64-bit microprocessor instruction-set architecture (ISA). The model is formalized in the logic of the ACL2 theorem prover [1], [2]. Our contribution is the formal specification and mechanical verification that our implementation provides a single, large, uniformly-addressed memory with space-efficient, high-speed (constant-time) performance. We desire high performance because of our interest in validating a (uni)processor model by simulating and comparing with expected results. As far as we know, our verified memory model is more time and space efficient than other models of a large memory formalized using the language of a theorem prover.

Microprocessor specifications require a model of its memory and its memory operations. Our model provides a memory of 2^{48} bytes; this is the address space defined by contemporary x86 implementations. Actually, some x86 implementations define a 52-bit address space, but such implementations require the use of the x86 memory management unit to access physical memory locations larger than 2^{48} bytes. If the need arises, we expect to be able to parameterize our model to offer larger (or smaller) memory address spaces.

Likely, every microprocessor design in the last 40 years has been modeled, and necessarily every such model includes a memory model, often written in C or Verilog. Our effort is focused on memory models that are (1) defined formally, (2) scale up to very large memories, (3) provide high-speed simulation, and (4) support mechanized reasoning. The memory model we present here defines four read (`rmXY`) operations and four write (`wmXY`) operations with the following interface signatures:

```
rm08: addr * mem → byte      wm08: addr * byte  * mem → mem
rm16: addr * mem → word      wm16: addr * word  * mem → mem
rm32: addr * mem → dword     wm32: addr * dword * mem → mem
rm64: addr * mem → qword     wm64: addr * qword * mem → mem
```

In this paper, we specify and verify a memory model satisfying the four (numbered) properties above and then we use this memory model to implement the eight memory functions just identified. In particular, Section V discusses classic read-over-write properties. Various microprocessor memory models can be layered on top of our memory model. Microprocessors providing virtual memory or other memory access mechanisms require a model of the physical memory; our focus here is the formalization of the physical memory interface. The complete source code and theorems for our memory model and its use in a partial x86 ISA specification may be found elsewhere [3].

Our efforts in this area started with the FM8501 and FM8502 microprocessors [4], [5], which included complete memory models whose performance was linear in the address size; thus, these models were not practical for simulating large memories. Our FM9001 microprocessor model [6] included a tree-based memory model that provided constant-time, tree-based accesses. Anthony Fox has developed a tree-based memory using HOL for his ARM microprocessor model, with a focus on program verification performance measured in tens of accesses per second [7]; by comparison, our performance is measured in hundreds of thousands of accesses per second (see Section VI).

Jared Davis used ACL2 to implement a tree-based 64-bit memory [8], which could make several hundred thousand accesses per second running on an Intel Pentium 4 in 2006. David Hardin (personal communication) reports 350,000 bytes/second on a 2.4 GHz Intel Core 2 Duo, using a version of Davis's model incorporated into an AAMP7 model [9]. The memory models of Davis and Hardin provide less than 1% of the memory performance we present here.

We begin by providing background on ACL2, the system that we are using for memory modeling. In Section III we present our two-level memory model. Section IV discusses invariants on our model and their role in efficient execution and fundamental properties. In Section V we present our higher-level read and write operations for bytes, words, doublewords, and quadwords, together with formally verified read-over-write properties of our memory model. Because we are using this memory model to support the modeling of microprocessor specifications, Section VI provides some memory access/update benchmark data. We conclude by observing that our memory implementation has been verified to operate correctly while providing sufficient performance to be used as the foundation of an ISA simulator.

II. ACL2 PRELIMINARIES

ACL2 [1] is a freely available system that provides a theorem prover and a programming language, both of which are based on a first-order logic of recursive functions [10], [11]. The logic is compatible with Common Lisp — indeed, "ACL2" is an acronym that might be written as "ACL^2" and stands for "A Computational Logic for Applicative Common Lisp" — and thus an executable image can be built on any of seven Common Lisp implementations. As a result, ACL2 provides efficient execution by way of Common Lisp compilers.

The initial theory for ACL2 contains axioms for primitive functions such as `car` (the head of a list or first component of a pair) and `cdr` (the tail of a list or second component of a pair). It also contains axioms for Common Lisp functions, such as `ash` (arithmetic shift), and it introduces axioms for user-supplied definitions.

ACL2 provides a top-level read-eval-print loop. Arbitrary ACL2 expressions may be submitted for evaluation. Of special interest are *events*, including definitions and theorems; these modify the logical database for subsequent proof and evaluation. For example, our memory model defines `n45p` to return true on 45-bit natural number inputs.

Links to numerous papers that apply ACL2, as well as detailed hypertext documentation and installation instructions, may be found on the ACL2 home page [2]. In the remainder of this section we briefly introduce aspects of ACL2 that are referenced in the remainder of this paper.

A. ACL2 basics

As is the case for Lisp, the syntax of ACL2 is generally case-insensitive and is based on prefix notation: (`function` `argument`$_1$ ... `argument`$_k$). For example, the term denoting the sum of `x` and `y` is `(+ x y)`. A semicolon ('`;`') begins a comment to the end of the line, generally shown in *italics* in this paper. Other ACL2 syntax used in this paper will probably make sense from the context, but we say a bit here about local variables, which may be introduced using `let` for parallel binding or `let*` for sequential binding. The term

```
(let ((x₁ t₁)
      (x₂ t₂)
      ...)
  (f ... x₁ ... x₂ ...))
```

binds variable x_1 to the value of term t_1, variable x_2 to the value of term t_2, and so on, before evaluating the indicated call of `f`. `Let*` is similar but has a sequential semantics: each binding applies to subsequent bindings. The following log illustrates the difference between the parallel bindings of `let` and the sequential bindings of `let*`.

```
ACL2 !>(let ((x 3))
         (let ((x (1+ x))  ; x is bound to 4
               (y x))      ; y is bound to old  x: 3
           (list x y)))  ; return list of  x and  y
(4 3)
ACL2 !>(let ((x 3))
         (let* ((x (1+ x)) ; x is bound to 4
                (y x))     ; y is bound to new  x: 4
           (list x y)))  ; return list of  x and  y
(4 4)
ACL2 !>
```

Functions in ACL2 may return multiple values. Logically, a multiple-value return is just a return value that is a list; but the implementation can avoid building list objects. Syntactic restrictions enforce proper use of multiple values. The primitives `mv` and `mv-let` create and bind multiple values, respectively, as we now illustrate (see [12] for details). The following function takes two numbers and uses the if-then-else primitive to return two values: the smaller and larger of those numbers, respectively. Note that here and throughout the paper, we avoid using the Lisp `defun` command, showing instead just the logical axiom added by the definition. For complete definitions, including `declare` forms that can improve efficiency and specify *guards* (*cf.* Section II-B), see the associated technical report [3].

```
Definition.
(min-max x y)
= (if (< x y) (mv x y) (mv y x))
```

The next function exponentiates the smaller of two numbers to the power of the larger.

```
Definition.
(expt-min-max x y)
= (mv-let (smaller bigger)
          (min-max x y)
          (expt smaller bigger))
```

Then for example:

```
ACL2 !>(expt-min-max 2 5)
32
ACL2 !>(expt-min-max 5 2)
32
ACL2 !>
```

B. Definitions and guards

The logic of ACL2 is untyped. However, ACL2 definitions may specify preconditions, known as *guards*. Consider for example the following definition of a function that returns the reciprocal of the difference of its inputs.

```
Definition.
(f x y)
= (/ (- x y))
Guard:
(and (rationalp x)
     (rationalp y)
     (not (equal x y)))
```

When this form is submitted, ACL2 performs *guard verification*, a static check (using the theorem prover) that for every function call that takes place during evaluation, the arguments satisfy the guard of that function. The example above generates the following two proof obligations, each under the hypothesis of the above guard: `x` and `y` are distinct rational numbers.

- The indicated subtraction requires that its arguments, `x` and `y`, are rationals.
- The indicated reciprocal operation requires that its argument, `(- x y)`, is a non-zero rational.

ACL2 easily discharges these proof obligations. Subsequently, any call of `f` will be evaluated in Common Lisp using the above code. Indeed, guards provide a link between the ACL2 logic and the host Lisp implementation, by allowing the use of Common Lisp evaluation in a way that avoids runtime errors.

Note that while guards are important for supporting evaluation by the host Lisp, they are irrelevant logically. For example, the ACL2 logic includes the following axiom, which implies that the reciprocal of a non-number or zero is zero.

```
Axiom. completion-of-unary-/
(equal (/ x)
       (if (and (acl2-numberp x)
                (not (equal x 0)))
           (/ x)
         0))
```

Thus, for example, one can prove `(equal (/ 0) 0)` with the ACL2 theorem prover. An attempt to evaluate `(/ 0)` (the reciprocal of zero) in the ACL2 read-eval-print loop will, by default, result in an error that reports a guard violation.

C. Single-threaded objects (stobjs)

Our memory model uses ACL2 single-threaded objects, or *stobjs* [13]. The first-order logic of ACL2 represents stobjs using linear lists, without side-effects. But for execution, ACL2 enforces syntactic *single-threadedness* restrictions on function definitions involving stobjs. so that they provide constant-time access and update using arrays, which can be made resizable. ACL2 provides detailed stobj documentation [12]; here we use an example to convey key ideas.

The following ACL2 event specifies a single-threaded object, `st`, which has a single field, `store`, which in turn is an array of 31-bit non-negative integers, initially all 0. Although `store` initially has length 8, it can be resized to arbitrary lengths.

```
(defstobj st
  (store :type (array (unsigned-byte 31) (8))
         :initially 0
         :resizable t))
```

Logically, `st` is just a one-element list whose unique element, `store`, is itself just a list. The following theorem makes this claim formally, where `store-length` is a function introduced by the above `defstobj` event, returning the number of entries in `store`. Throughout this paper and also in our ACL2 development, we give theorems descriptive names.

```
Theorem. store-length-computes-len
(implies
 (stp st)  ; st satisfies its recognizer
 (and (consp st)            ; st is a list
      (null (cdr st))       ; st has only one member
      (equal (len (car st)) ; list-length of  store
             (store-length st))))
```

However, the implementation guarantees that no list construction is performed when updating `store`, and unlike linear list operations, every access to `store` is done in constant time with an array indexing operation.

D. About proofs

Our presentation below focuses on formalization and proof highlights, avoiding proof details. Full ACL2 input scripts may be found elsewhere [3].

Our proofs of the read-over-write lemmas in Section V-B take advantage of the GL symbolic simulation package [14]. That package requires the experimental "hons" extension, ACL2(h), of the ACL2 theorem prover [15], [12], which we therefore used for this effort.

III. Memory Structure, Access, and Update

Our memory model is based on an array of 64-bit quadwords, providing the illusion of a memory containing 2^{48} bytes. The model includes read and write operations, `memi` and `!memi`, for quadwords (64 bits). Later, in Section V, we build on these primitives to define byte-addressed reads and writes for various sizes: byte (8 bits), word (16 bits), doubleword (32 bits), and quadword (64 bits).

The correctness of our model is captured by the following standard property of arrays. We briefly discuss its proof in Subsection IV-C.

```
Theorem. memi-!memi
(implies
  (and (x86-64p x86-64)        ; Memory OK
       (n45p i)                ; Read address OK
       (n45p j))               ; Write address OK
  (equal (memi i               ; Read address
               (!memi j        ; Write address
                      v        ; Value to write
                      x86-64)) ; Initial memory
         (if (equal i j)       ; For equal addresses
             v                 ; the read value is v
           (memi i x86-64))))) ; else, unchanged
```

Our memory model is implemented using a data structure with three fields; see Fig. 1. Although the memory is conceptually an array of 2^{48} bytes, we choose our data structure for space efficiency. Our choice of 27 bits is somewhat arbitrary, but intended to balance the size of `mem-table` — 2^{27} (134M) entries — with the size of the `mem-array`, which initially consists of (somewhat arbitrarily) 100 pages, each containing 2^{21} bytes (2MB).

- The memory address table, `mem-table`, is indexed by the top (most significant) 27 bits of a 45-bit quadword address. Its valid entries are 45-bit addresses.
- The memory array, `mem-array`, is indexed by 45-bit quadword addresses from `mem-table`. Its entries are the memory quadword values.
- A 45-bit quadword address, `mem-array-next-addr`, points to the next free two-megabyte section ("page") of `mem-array`.

We use the ACL2 *stobj* mechanism (see Section II) to implement these fields.

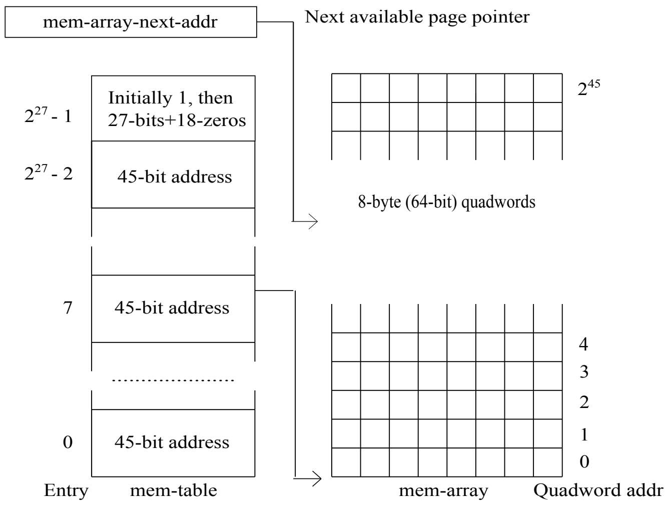


Fig. 1. Memory System

```
(defstobj x86-64
 ; some fields elided
  (mem-table
   :type
   (array (unsigned-byte 45)
          (*mem-table-size*))  ; 2^27
   ;; either 1 or a memory page address
   :initially 1
   :resizable nil)
  (mem-array  ; resizable array of quadwords
   :type (array (unsigned-byte 64)
                (*initial-mem-array-length*))
   :initially 0
   :resizable t)
  (mem-array-next-addr
   :type  ; natural number < 2^45
   (integer 0 35184372088832)
   :initially 0)
)
```

The first field defines mem-table as an array of 2^{27} entries where each entry is constrained to be a 45-bit natural number, initially 1. The second field defines a memory array of 2^{45}, unsigned 64-bit integers (quadwords), with its initial entries all being 0. This array has (an initial length of) *initial-mem-array-length* entries (arbitrarily set to $100 * 2^{18}$); but since it is declared resizable, it will be extended automatically as necessary. The third field, a 45-bit integer named mem-array-next-addr, tracks the space allocated in mem-array.

We initialize mem-table values to 1 so we can distinguish which memory table entries are valid. The valid entries in mem-table are unique 45-bit addresses that are aligned to two-megabyte boundaries; that is, the bottom (least significant) 18 bits of these addresses are all zero. This choice results in each mem-table entry pointing to the start of a two megabyte "page" in mem-array. In our implementation, mem-array is initially allocated an amount of memory corresponding to a positive integral number of two-megabyte pages. When the demand for memory exceeds the available memory pages, mem-array is dynamically extended (until the underlying operating system fails to be able to allocate memory).

Our memory implementation writes to mem-table whenever a write is presented for which the corresponding two-megabyte page has no entry in mem-table, following a process that can be thought of as a one-level paging scheme. Suppose for example that we start with an empty memory and perform three writes, as shown below.

- **First write to memory: at quadword address** $7 * 2^{18} + 345$. (See Fig. 1.) The corresponding page index into mem-table is 7, selected by right shifting the quadword address by 18 bits. Since this is the first write, our memory write function will see that mem-table has value 1 at index 7, indicating that the corresponding two-megabyte page has no index in mem-table. Index 7 will then obtain the value of mem-array-next-addr, $0 * 2^{18}$, which corresponds to the first available "page" in mem-table. Also, mem-array-next-addr is bumped up to the next page address, $1 * 2^{18}$. An address into mem-array is then constructed by combining the page address of $0 * 2^{18}$ with the original low 18 address bits, in this case 345, to obtain $0 * 2^{18} + 345$. We write the given quadword data to that address of mem-array.
- **Second write to memory: at quadword address** $23 * 2^{18} + 12$. Following the steps above, we find an invalid entry at index 23, which we replace by the current value of mem-array-next-addr, $1 * 2^{18}$. (And, mem-array-next-addr is then bumped up by 2^{18}, to $2 * 2^{18}$.) We then write the quadword data into mem-array at index $1 * 2^{18} + 12$.
- **Third write to memory: at quadword address** $7*2^{18}+5$. This time we find a valid entry in mem-table, namely at index 7 as placed by the first write. So we write the quadword data into mem-array at index $0 * 2^{18} + 5$.

In summary, a mem-table entry for the top 27 bits of an address serves as the base index for the address where we will write a quadword to mem-array. The full index for writing into mem-array is the sum (performed by the logical inclusive 'or' function logior) of the base index and the bottom 18 bits of the original address. We expect that it would be easy to remove 17 of those 18 bits, leaving just one "valid" bit, and we may do that in the future; but we liked the simplicity of using logior.

The same addressing scheme is used when reading, but mem-array is never extended on reads. If there is an appropriate mem-table entry, then the value returned will be found in mem-array using the scheme described above. Otherwise, the default value 0 is returned.

Our primitive memory read and write functions, memi and !memi, are defined as described above. In particular, note that !memi calls a function add-page when necessary to extend the available memory and obtain a mem-array address from mem-array-next-addr.

```
Definition.
(memi i x86-64)
= (let* ((i-top27  ; right shift 18 bits
          (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (if (eql addr 1)  ; page is not present
        *default-mem-value*
      (let ((index (logior addr
                           (logand #x3ffff i))))
        (mem-arrayi index x86-64))))
```

```
Definition.
(!memi i v x86-64)
= (let* ((i-top27 (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (mv-let
     (addr x86-64)
     (if (eql addr 1)  ; if page is not present
         (add-page i-top27 x86-64)  ; add a page
       (mv addr x86-64))
     (!mem-arrayi (logior addr (logand #x3ffff i))
                  v
                  x86-64)))
```

```
Definition.
(add-page i x86-64)
= (let* ((addr (mem-array-next-addr x86-64))
         (len (mem-array-length x86-64))
         (x86-64
          (if (eql addr len)  ; must resize!
            (resize-mem-array
             (min (* *mem-array-resize-factor* len)
                  *2^45*)
             x86-64)
            x86-64))
         (x86-64  ; Add next new page.
                 (!mem-array-next-addr
                  (+ addr *2^18*)
                  x86-64))
         (x86-64 (!mem-tablei i addr x86-64)))
    (mv addr x86-64))
```

The question remains of whether this scheme always works. The next section addresses this question.

IV. Memory invariant and its consequences

In this section we introduce our invariant on the memory. We then sketch how it supports proofs of properties that support efficient execution. Finally, we show how our invariant supports the proof of the key property of our memory model.

A. The memory invariant

Recall our two-level memory, where `mem-table` is an array that contains 2^{18}-aligned addresses indexing into `mem-array`, a resizable array containing 64-bit data, where those addresses are below the (2^{18}-aligned) address limit, `mem-array-next-addr`. Our invariant, stated informally below, incorporates these and other properties. We write `table-max-index` to denote the maximum index into `mem-table`, i.e., one less than the length of `mem-table`, and we write `mem-array-length` to denote the current length of `mem-array`.

1) `mem-array-next-addr` $\leq$ `mem-array-length`.
2) `*initial-mem-array-length*` $\leq$ `mem-array-length`.
3) `#x3ffff` & `mem-array-length` = 0, i.e., `mem-array-length` is 2^{18}-aligned.
4) `mem-array-next-addr` = $2^{18} * k$, where k is the number of valid entries in `mem-table` (entries not equal to 1).
5) Every valid entry in `mem-table` is 2^{18}-aligned and is less than `mem-array-next-addr`.
6) There are no duplicate valid entries in `mem-table`.
7) The value is 0 in `mem-array` at every index at or exceeding `mem-array-next-addr`.

The function `good-memp` formalizes our memory invariant, as described informally by these seven clauses. The invariant on our stobj is the conjunction of basic structural properties, represented by the stobj recognizer `x86-64p-pre`, and our memory invariant.

```
Definition.
(x86-64p x86-64)
= (and (x86-64p-pre x86-64)
       (good-memp x86-64))
```

The following theorem formalizes invariance for our basic memory write operation. We have also proved such theorems for the higher-level memory write operations presented in Section V.

```
Theorem. x86-64p-!memi
(implies (and (x86-64p x86-64) (n45p i) (n64p v))
         (x86-64p (!memi i v x86-64))))
```

B. Guard verification using the invariant

Section II discussed the role of *guards* in supporting efficient execution. In this section we illustrate the important role played by our invariant for verifying guards, using as a key example our basic memory read function, `memi`.

Recall that `memi` reads the quadword at address `i` from the memory of our `x86-64` stobj. Its guard is given as follows.

```
(and (n45p i)  ; 45-bit quadword address
     (x86-64p x86-64))
```

The interesting case for reading a 45-bit quadword address is that its top 27 bits index into a valid entry of `mem-table`, which is an index into `mem-array`. A corresponding proof obligation arises from guard verification for function `memi`; it states that the corresponding index into `mem-array` is in bounds.

```
(implies
 (and (x86-64p x86-64)
      (< i 2^45)
      (<= 0 i)
      (integerp i)
 ; The next conjunct says that we have a valid
 ; mem-table entry.
      (not (equal (nth (ash i -18)
                       (nth *mem-tablei* x86-64))
                  1)))
 ; We conclude that the index into  mem-array,
 ; as represented by the  logior call below, is
 ; less than the length of  mem-array.
 (< (logior (logand #x3ffff i) ; low 18 bits of i
            (nth (ash i -18)   ; top 27 bits of i
                 (nth *mem-tablei* x86-64)))
    (len (nth *mem-arrayi* x86-64))))
```

In order for this formula to be a theorem, the hypothesis `(x86-64p x86-64)` must be sufficiently strong. We have checked mechanically with ACL2 that this is indeed the case.

C. A fundamental read-over-write lemma

Recall the key property `memi-!memi` from the start of Section III, which characterizes the effect of a quadword write on the memory. It is crucial for proving analogous properties of higher-level read and write functions, as discussed in Section V. That property naturally breaks into two lemmas. One of those is the following, for the case that the address for reading is the same as the address that is written.

```
Theorem. memi-!memi-same
(implies (x86-64p x86-64)
         (equal (memi i (!memi i v x86-64))
                v))
```

With suitable hints and lemmas, ACL2 proves this theorem. But among the lemmas applied in its proof, as reported by the prover, the following is one that is critical, as the proof fails without it. Note that it corresponds to Clause 4 of our invariant.

```
Theorem. logand-mem-array-next-addr
(implies (good-memp x86-64)
         (equal (logand #x3ffff
                        (nth *mem-array-next-addr*
                             x86-64))
                0))
```

We now consider the other case, that is, where the addresses are distinct.

```
Theorem. memi-!memi-different
(implies (and (not (equal i j))
              (n45p i)
              (n45p j)
              (x86-64p x86-64))
         (equal (memi i (!memi j v x86-64))
                (memi i x86-64)))
```

Consider what happens if two mem-table indices contain the same value. For example, suppose that quadword addresses `i` and `j` are 0 and 2^{18}, respectively, yet the corresponding mem-table entries at indices 0 and 1 both have value 0. Also suppose that all values stored in `mem-array` are 0, and that the value `v` is 1. Then, even though `i` and `j` are distinct, the equality displayed above is false, as `(memi i (!memi j v x86-64))` is 1 yet `(memi i x86-64)` is 0.

It is here that the memory invariant saves us. Specifically, Clause 6 prohibits duplicate entries in `mem-table`. We prove that every operation on the memory preserves the memory invariant.

V. Using the Memory Model: Reads and Writes

We have seen functions `memi` and `!memi` for reading and writing quadwords (8-byte natural numbers) at quadword-aligned memory addresses. But for our intended application of modeling the x86 ISA [3], we also require functions that read and write bytes, words, doublewords, and quadwords at arbitrary addresses. We tour those below and then discuss theorems relating reads and writes.

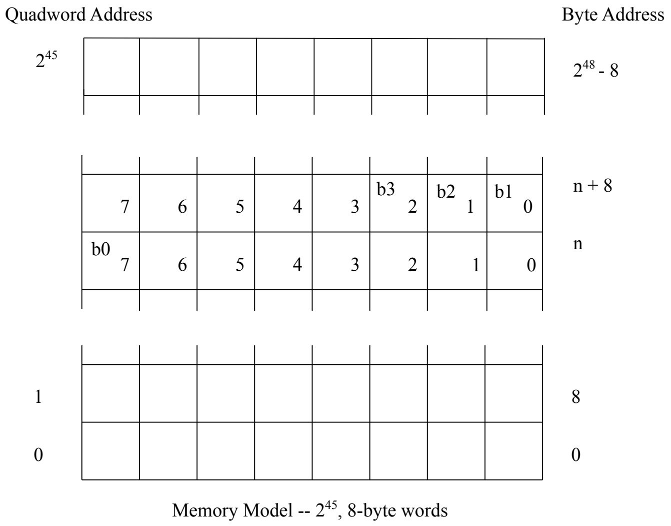


Fig. 2. Misaligned memory access

For byte reads the memory is read once. But for non-aligned accesses of more than one byte, it may be necessary to read the memory twice because the access may be split across two (64-bit) quadwords. This possibility is illustrated in Fig. 2 for a doubleword (four bytes: b0, b1, b2, b3) stored at address `n+7`. Here, `n` is the address of a byte on a quadword boundary (`n` is a multiple of 8). We use little-endian format, which requires that the least significant byte appear at address `n+7`, the next byte at `n+8`, the next byte at `n+9`, and the most significant byte at `n+10`.

A. Read and write functions

We implement byte, word, doubleword, and quadword read and write operations using the primitive quadword memory-read and memory-write functions, `memi` and `!memi`. The following function reads a single byte from memory.

```
Definition.
(rm08 addr x86-64)
= (let* ((byte-num     (n03 addr))
         (qword-addr   (ash addr -3))
         (qword        (memi qword-addr x86-64))
         (shift-amount (ash byte-num 3))
         (shifted-qword (ash qword
                             (- shift-amount))))
    (n08 shifted-qword))
Guard:[1]
(and (n48p addr)
     (x86-64p x86-64))
```

The following lemma is critical in order to verify guards for the above function. Specifically, it is used in the proof of the guard obligation from the definition of `rm08` for the call `(ash qword (- shift-amount))`, which states that `qword` is an integer, where `qword` is `(memi qword-addr x86-64)`.

[1]We show a formula that is logically equivalent to the guard.

```
Theorem. memi-is-unsigned-byte-64
(implies (and (x86-64p x86-64)
              (n45p addr))
         (n64p (memi addr x86-64)))
```

But why does this lemma hold? Clause 5 of our invariant (Section IV-A) is crucial, and is a consequence of hypothesis `(x86-64p x86-64)`:

> Every valid entry in `mem-table` is 2^{18}-aligned and is less than `mem-array-next-addr` (Fig.??).

Indeed, if we tell ACL2 to ignore ("disable") two rewrite rules corresponding to this property, the proof fails.

We turn now from reading to writing a byte.

```
Definition.
(wm08 addr byte x86-64)
= (let* ((byte-num        (n03 addr))
         (qword-addr      (ash addr -3))
         (qword           (memi qword-addr x86-64))
         (shift-amount    (ash byte-num 3))
         (byte-mask       (ash #xff shift-amount))
         (qword-masked    (logand (lognot byte-mask)
                                  qword))
         (byte-to-write   (ash byte shift-amount))
         (qword-to-write  (logior qword-masked
                                  byte-to-write)))
    (!memi qword-addr qword-to-write x86-64))
```

It will be important to maintain our invariant after doing a write, so that the guards (which include our invariant) are met for subsequent memory operations. We therefore prove the following lemma.

```
Theorem. x86-64p-wm08
(implies (and (x86-64p x86-64)
              (n48p addr)
              (n08p byte))
         (x86-64p (wm08 addr byte x86-64)))
```

Reads and writes of more than one byte are built up in layers. For example, here is the function for reading four bytes, which invokes the two-byte read function when the addresses cross a quadword boundary. Notice that the call of `n48p` in the guard leaves room to read four bytes.

```
Definition.
(rm32 addr x86-64)
= (let ((byte-num (n03 addr)))
    (cond
     ((<= byte-num 4)
      (let* ((qword-addr   (ash addr -3))
             (qword        (memi qword-addr x86-64))
             (shift-amount (ash byte-num 3))
             (shifted-qword (ash qword
                                 (- shift-amount))))
        (n32 shifted-qword)))
     (t  ; byte-num is 5, 6, or 7
      (let* ((word0 (rm16 addr x86-64))
             (word1 (rm16 (n48+! 2 addr) x86-64)))
        (logior (ash word1 16) word0)))))
```

B. Read-over-write theorems

The following theorem characterizes the effect of reading a byte from address `i` after writing a byte, `v`, at address `j`. The result, of course, is `v` if `i` equals `j`; otherwise the write does not affect the value returned by the read. The proof relies on the lemma `memi-!memi`, which takes advantage of the invariant, `(x86-64p x86-64)`; see Section IV-C.

```
Theorem. rm08-wm08
(implies (and (x86-64p x86-64)
              (n48p i) (n48p j) (n08p v))
         (equal (rm08 i (wm08 j v x86-64))
                (if (equal i j)
                    v
                  (rm08 i x86-64))))
```

The corresponding lemma for two bytes is a bit more complex, as the cases for the resulting read depend on how the two address regions overlap.

```
Theorem. rm16-wm16
(implies
 (and (x86-64p x86-64)
      (natp i) (n48p (1+ i))
      (natp j) (n48p (1+ j))
      (n16p v))
 (equal (rm16 i (wm16 j v x86-64))
        (cond ((equal i j)
               v)
              ((equal j (1+ i))
               (logior (* *2^8* (logand #xff v))
                       (rm08 i x86-64)))
              ((equal i (1+ j))
               (logior (ash (logand #xff00 v) -8)
                       (* *2^8*
                          (rm08 (+ 1 i) x86-64))))
              (t
               (rm16 i x86-64)))))
```

Our approach to proving this theorem is to reduce it to the preceding theorem for single-byte reads and writes. Thus, we characterize two-byte reads in terms of single-byte reads, and similarly for writes. Here is the relevant lemma for writes.

```
Theorem. wm16-as-wm08
(implies
 (and (x86-64p x86-64)
      (natp addr)
      (n48p (1+ addr))
      (n16p word))
 (equal (wm16 addr word x86-64)
        (let* ((x86-64
                (wm08 addr
                      (logand word #xff)
                      x86-64))
               (x86-64
                (wm08 (+ 1 addr)
                      (ash (logand word #xff00) -8)
                      x86-64)))
          x86-64)))
```

Recall that `wm08` is defined in terms of the primitive quadword write operation, `!memi`. Since the proof of the lemma above involves reasoning about successive writes, it is not surprising that the following is critical for its proof.

```
Theorem. !memi-!memi-same
(implies (x86-64p x86-64)
         (equal (!memi addr v1
                       (!memi addr v2 x86-64))
                (!memi addr v1 x86-64)))
```

VI. Efficient execution

We have fabricated some memory-copy tests to get an idea of the performance of our memory implementation. The Lisp runs reported below used a 3.5 GHz Intel Xeon processor.

```
(defun copy (from to count x86-64)
  (declare (type (unsigned-byte 29) count)
           (type (unsigned-byte 45) from to)
           (xargs :guard
                  (and (< (+ from count) *2^45*)
                       (< (+ to count) *2^45*)
                       (x86-64p x86-64))
                  :stobjs (x86-64)))
  (if (zpf count)
      x86-64
    (let* ((value (memi from x86-64))
           (x86-64 (!memi to value x86-64)))
      (copy (1+ from) (1+ to) (1- count) x86-64))))
```

Function `copy-test`, called below, writes a 1 at each address below its first argument, `addr`, and then calls `(copy 0 addr addr x86-64)`. Note that the first two runs copy 1 GB (either 128 quadwords copied 1M times or 128K quadwords copied 1K times), while the third copies 1 GB (128M quadwords) ten times, to amortize the memory initialization with ones.

```
(time$  ; 2.9 seconds
 (copy-test 128             (* 1024 1024) x86-64))

(time$  ; 2.8 seconds
 (copy-test (* 128 1024)      1024          x86-64))

(time$  ; 29.9 seconds (for 10x memory ops)
 (copy-test (* 128 1024 1024) 10            x86-64))
```

The copying of approximately 350M bytes/second corresponds to 700M memory byte accesses per second, which we find encouraging. However, this is slower by about a factor of 9 than the analogous three runs of a corresponding C program compiled with `gcc -O3`, with times of 0.330 seconds, 0.320 seconds, and 3.260 seconds, respectively.

VII. conclusion

Our formal memory model has been proven to provide the illusion of a complete 2^{48}-byte memory. Our implementation provides time- and space-efficient, constant-time memory read/write operations, thus supporting validation of ISA simulators. We represent our large memory by using an expandable collection of pages indexed by a table of page pointers, so that we can provide the illusion of having a memory containing 2^{48} bytes. We have assured that our memory model is correct throughout the entire address range by proving requisite properties of our model. This kind of modeling is important for large-scale memory systems as they cannot be practically built nor tested for all manner of configurations.

Our memory model permits 350M bytes per second to be copied from one part of memory to another part, which we believe exceeds the performance of all other theorem-prover-based memory models for such large address spaces. When used within our evolving x86 microprocessor ISA specification, our model provides sufficient performance to allow binary code to be executed at more than 500,000 instructions per second [3]. We expect to use this or a similar memory model as we move forward with our microprocessor modeling efforts [16].

Acknowledgments

This material is based upon work supported by DARPA under contract number N66001-10-2-4087 and by ForrestHunt, Inc. We thank Jared Davis, Shilpi Goel, Sandip Ray, Anna Slobodova, and Sol Swords for useful feedback on drafts of this paper. We also thank Shilpi Goel for drawing the figures and David Hardin for providing performance data on an AAMP7 memory model. Finally, we thank the referees for providing useful feedback on our submission.

References

[1] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Boston, MA: Kluwer Academic Publishers, Jun. 2000.

[2] M. Kaufmann and J. S. Moore, "ACL2 home page," see URL http://www.cs.utexas.edu/users/moore/acl2.

[3] M. Kaufmann and W. A. Hunt, Jr., "Towards a formal model of the x86 ISA," Department of Computer Sciences, University of Texas at Austin, Tech. Rep. TR-12-07, May 2012.

[4] W. A. Hunt, Jr., *FM8501: A Verified Microprocessor*, ser. LNAI. Springer-Verlag, 1994, vol. 795.

[5] ——, "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, pp. 429–460, 1989.

[6] W. A. Hunt, Jr. and B. Brock, "A Formal HDL and Its Use in the FM9001 Verification," in *Mechanized Reasoning and Hardware Design*, ser. Prentice-Hall International Series in Computer Science, C. A. R. Hoare and M. J. C. Gordon, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1992, pp. 35–48.

[7] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *ITP 2010*, ser. LNCS, no. 6172. Springer, 2010.

[8] J. C. Davis, "Memories: Array-like records for ACL2," in *Proceeding of the* 6th International Workshop on the ACL2 Theorem Prover and its Applications, no. ISBN: 0-9788493-0-2. ACM, 2006.

[9] D. Hardin, E. W. Smith, and W. D. Young, "A Robust Machine Code Proof Framework for Highly Secure Applications," in *Proceedings of the* 6*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, P. Manolios and M. Wilding, Eds. ACM, Jul. 2006, pp. 11–20.

[10] M. Kaufmann and J. S. Moore, "Structured Theory Development for a Mechanized Logic," *Journal of Automated Reasoning*, vol. 26, no. 2, pp. 161–203, 2001.

[11] ——, "A Precise Description of the ACL2 Logic," 1997, see URL http://www.cs.utexas.edu/users/moore/publications/km97.ps.gz.

[12] ——, "ACL2 documentation," see URL http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html.

[13] R. S. Boyer and J. S. Moore, "Single-threaded Objects in ACL2," in *Practical Aspects of Declarative Languages (PADL)*, ser. LNCS, S. Krishnamurthy and C. R. Ramakrishnan, Eds., vol. 2257. Springer-Verlag, 2002, pp. 9–27.

[14] S. Swords and J. Davis, "Bit-blasting ACL2 theorems," in *Proceeding 10th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, D. Hardin and J. Schmaltz, Eds., vol. 70, 2011, pp. 84–102.

[15] R. S. Boyer and W. A. Hunt, Jr., "Function Memoization and Unique Object Representation for ACL2 Functions," in *Proceedings of the* 6*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*. ACM, Aug. 2006, pp. 81–89.

[16] S. Goel, W. A. Hunt, Jr., and M. Kaufmann, "Towards an ACL2 model of the x86 ISA," in preparation.

Verification with Small and Short Worlds

Rohit Sinha, UC Berkeley
Cynthia Sturton, UC Berkeley
Petros Maniatis, Intel Labs
Sanjit A. Seshia, UC Berkeley
David Wagner, UC Berkeley

***Abstract*—We consider the verification of safety properties in systems with large arrays and data structures. Such systems are common at the low levels of software stacks; examples are hypervisors and CPU emulators. The very large data structures in such systems (e.g., address-translation tables and other caches) make automated verification based on straightforward state-space exploration infeasible. We present S^{2}W, a new abstraction-based model-checking methodology to facilitate automated verification of such systems. As a first step, inductive invariant checking is performed. If that fails, we compute an abstraction of the original system by precisely modeling only a subset of state variables while allowing the rest of the state to evolve arbitrarily at each step. This subset of the state constitutes a "small world" hypothesis, and is extracted from the property. Finally, we verify the safety property on the abstract model using bounded model checking. We ensure the verification is sound by first computing a bound on the reachability diameter of the abstract model. For this computation, we developed a set of heuristics that we term the "short world" approach. We present several case studies, including verification of the address translation logic in the Bochs x86 emulator, and verification of security properties of several hypervisor models.**

I. Introduction

CPU emulators lie in the foundational layer of much of today's computing infrastructure: CPU emulation is used by virtualization software (hypervisors and virtual machine monitors) in both testing and production to secure, analyze, and multiplex critical systems [7], [15], [28]. Unfortunately, although critical, CPU emulators are not often verified, and are frequently found incorrect [19], [20] or – worse – vulnerable to attacks [4].

A particular challenge for the verification of CPU emulators and hypervisors is their use of large data structures. For example, logical-to-physical address translation requires data structures to store the CPU's Translation Look-aside Buffer (TLB) and page tables. While these structures are finite-length for any given processor, they are usually too large to represent precisely for verification; often, they are abstracted to be of unbounded length. The data structures in the resulting model of the system are thus *parametrized*: the indices into those structures are *parameters*, taking values in a very large or even infinite domain (typically finite-precision bit-vectors or the integers). The techniques proposed for verifying such parametrized systems fall into two classes: those based on a small-model or cut-off theorem (e.g., [11], [12], [24]), or those based on abstraction (e.g., [8], [14], [18]). While existing approaches are elegant and effective for their respective problem domains, they fall short for the problems we consider: the small-model approaches usually restrict expressiveness, while abstraction-based approaches either focus on control properties (as opposed to equivalence/refinement) or handle only certain kinds of data structures. In both cases, some of the realistic case studies we consider cannot be handled. (We make a fuller comparison in Sec. VI.)

In this paper, we present a new semi-automatic methodology for verifying safety properties in systems with large data structures. Our approach comprises three steps. First, we employ standard (mathematical) induction to verify the safety property, and if that succeeds, the process is complete. Second, if induction fails, we create an over-approximate abstraction of the system, the "small world," in which unbounded data structures are parametrized and, in general, only a subset of the state is updated as per the original transition relation (e.g., only a few entries of the unbounded data structures); the rest of the state is updated with arbitrary values at each step. With this abstraction, the model is more amenable to state-space exploration. Third, we attempt to find a bound k on the reachability diameter of the small world so that, if bounded model checking for k steps succeeds in the small world, then the safety property must hold in the small world, and since that is an over-approximation of the original system model, then the safety property holds there as well. Heuristics are presented for finding k that are effective for the class of systems we consider. We term this BMC-based approach the "short world" method, since it relies on computing a "short" bound for BMC. Our overall approach, termed *Small-Short-World (S^{2}W)*, is implemented on top of the UCLID system [9], which verifies abstract term-level models using satisfiability modulo theories (SMT) solving. Note that the temporal safety verification problem for our class of systems is undecidable. As a result, S^{2}W is a semi-decision procedure.

In summary, the novel contributions of this paper include:

- A semi-automatic procedure, S^{2}W, for verifying systems with large or unbounded data structures, using a combination of induction and abstraction-based model checking. The key new ideas are a set of heuristics for creating an abstract model and computing a bound on the reachability diameter of its state space.
- An extensive evaluation of S^{2}W on a wide range of CPU emulator and hypervisor models, proving safety properties critical to the security and correctness of those systems. Our examples include the TLB implementation in the Bochs x86 emulator [23] and a shadow page table system.

II. Running Example

We introduce here a running example: a simple read-only memory system with a single-entry cache. We prove an invariant about the value returned by a read command. We build a model in our modeling language and demonstrate the verification of the safety property using S^{2}W. Our example is meant to be small, understandable, and illustrative, rather than "real-world."

Our example system (Fig. 1) takes only one command, *read*, with a single parameter, the 32-bit address to be read; it returns a single-bit data value. At each read command, the cache is

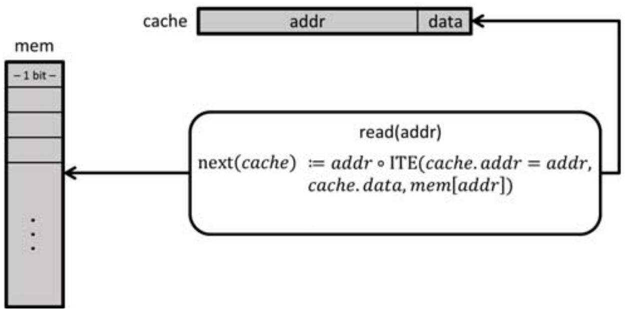

cache
addr
data
mem
– 1 bit –
read(addr)
next(cache) := addr ∘ ITE(cache.addr = addr, cache.data, mem[addr])

Fig. 1: An illustration of our running example: A read-only memory and a single-entry cache. The cache is updated on each read command.

first checked. If the cache contains the data for the address requested, that value is returned. Otherwise, the value is read from memory. In either case, the cache is updated with the requested address and the returned data value. The update to cache is shown in the above figure (we use "∘" to mean concatenation). We prove an invariant about the cache: if the cache holds a valid address, then the cached data value is equal to the value stored in memory at that address. In other words, we show that the cache is correct.

III. Formal Description of Problem

A. Notation and Terminology

A system is modeled as a tuple $\mathcal{S}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, $Init$, $\mathcal{A}$) where

- $\mathcal{I}$ is a finite set of input variables;
- $\mathcal{O}$ is a finite set of output variables;
- $\mathcal{V}$ is a finite set of state variables;
- $Init$ is a set of initial states, and
- $\mathcal{A}$ is a finite set of assignments to variables in $\mathcal{V}$. Assignments define how state variables are updated, and thus define the transition relation of the system.

Input and output variables are assumed combinational (stateless), without loss of generality. $\mathcal{V}$ is the only set of state-holding variables. Variables can be of two types: *primitives*, such as Boolean or bit-vector; and *memories*, which includes arrays, content-addressable memories (CAMs), and tables. An output variable is a function of $\mathcal{V} \uplus \mathcal{I}$. When representing a system without outputs, we will omit $\mathcal{O}$ from the representation. The set of initial states, $Init$, can either be viewed as a symbolic vector of terms representing any initial state, or as a Boolean-valued function of $\mathcal{V}$, written $Init(\mathcal{V})$.

Fig. 2 denotes the grammar for expressions in our modeling language. The language has three expression types: Boolean, bit-vector, and memory.

$$
\begin{array}{lll}
bE & ::= \mathbf{true} \mid \mathbf{false} \mid b \mid \neg bE \mid bE_1 \vee bE_2 & \\
 & \mid bE_1 \wedge bE_2 \mid bvE_1 = bvE_2 \mid \mathtt{bvrel}(bvE_1, \ldots, bvE_k) & (k \geq 1) \\
 & \mid UP(bvE_1, \ldots, bvE_k) \quad (k \geq 0) & \\
bvE & ::= c \mid v \mid \mathtt{ITE}(bE, bvE_1, bvE_2) & \\
 & \mid \mathtt{bvop}(bvE_1, \ldots, bvE_k) \ (k \geq 1) & \\
 & \mid mE(bvE_1, \ldots, bvE_l) \mid UF(bvE_1, \ldots, bvE_k) & (l \geq 1, k \geq 0) \\
mE & ::= A \mid M \mid \lambda(x_1, \ldots, x_k).bvE \quad (k \geq 0) &
\end{array}
$$

Fig. 2: Expression Syntax. c and v denote a bit-vector constant and variable, respectively, and b is a Boolean variable. bvop denotes any arithmetic/bitwise operator mapping bit vectors to bit vectors, while bvrel is a relational operator other than equality mapping bit vectors to a Boolean value. UF and UP denote an uninterpreted function and predicate symbol respectively. A and M denote constant and variable memories. $x_1, \ldots, x_k$ denote parameters (typically indices into memories) that appear in *bvE*.

The simplest Boolean expressions (*bE*) are the constants **true** and **false** or Boolean variables; more complicated expressions can be constructed using standard Boolean operators or using relational operators amongst bit-vector expressions. We also allow a Boolean expression to be an application of an uninterpreted predicate to bit-vector expressions.

Bit-vector expressions (*bvE*) include bit-vector constants, variables, if-then-else expressions (ITE), and expressions constructed using standard bit-vector arithmetic and bitwise operations. Additionally, bit-vector expressions can be constructed as applications of uninterpreted functions returning bit-vector values and applications of memories to bit-vector arguments. Each bit-vector expression has an associated bitwidth.

Finally, the primitive memory expressions (*mE*) can be (symbolic) constants or variables. More complex memory expressions can be modeled using the Lambda notation introduced by Bryant et al. [9] for term-level modeling; this includes the standard **write** (**store**) primitive for modeling arrays, as well as more general operations such as parallel updates to arrays, operations on CAMs, queues, and other data structures.

In addition to the above expressions, we will use the wildcard "$*$" to denote an arbitrary value of the appropriate type; it is used primarily to express non-determinism in the state update.

A *next-state assignment* α denotes assignment to a state variable and is a rule of the form $\mathtt{next}(x) := e$ or $\mathtt{next}(x) := \{e_1, e_2, \ldots, e_n\}$, where x is a signal in $\mathcal{V}$, and $e, e_1, e_2, \ldots, e_n$ are expressions that are a function of $\mathcal{V} \uplus \mathcal{I}$. The curly braces and "$*$" express non-deterministic choice. The set of all next-state assignments defines the *transition relation* $\mathcal{R}$ of the system. Formally, $\mathcal{R} = \bigwedge_{\alpha \in \mathcal{A}} r(\alpha)$, where $r(\mathtt{next}(x) := e) \doteq (x' = e)$ and $r(\mathtt{next}(x) := \{e_1, e_2, \ldots, e_n\}) \doteq \bigvee_{i=1}^{n}(x' = e_i)$, where x' denotes the next-state version of variable x. The wildcard "$*$" is translated at each transition into a fresh symbolic constant of the appropriate type. We will sometimes write the transition relation as $\mathcal{R}(\mathcal{V}, \mathcal{I}, \mathcal{V}')$ to emphasize that it relates current-state variables $\mathcal{V}$ and next-state variables $\mathcal{V}'$ based on the inputs $\mathcal{I}$ received.

Example 1: We formally describe our model from Sec. II. Let $\mathcal{S}_T$ = ($\mathcal{I}$, $\mathcal{O}$,$\mathcal{V}$, $Init$, $\mathcal{A}$) be the system, with

- $\mathcal{I} = \{addr\}$. *addr* is the 32-bit address to read from memory.
- $\mathcal{O} = \{out\}$. *out* is the value read from either memory or the cache.
- $\mathcal{V} = \{mem, cache\}$. *mem* is constant and is modeled as an array of (one-bit) bit-vectors. It is represented by an uninterpreted function that maps a 32-bit address to a single bit. *cache* is a 33-bit vector; it holds the one-bit data value and 32-bit address of that value.
- $Init = (mem_0, cache_0)$. *mem* is initialized to hold arbitrary data values at each address. *cache* is initialized to hold an invalid address, 0x00000000, with an arbitrary data value.
- $\mathcal{A}$. On each read command *cache* is updated with the address read and the value returned by the read; *mem* remains constant.

B. Problem Definition

Consider a system $\mathcal{S}$ modeled as described in the preceding section. We similarly model the environment $\mathcal{E}$ that provides the inputs for $\mathcal{S}$ and consumes its outputs. The composition of $\mathcal{S}$ and $\mathcal{E}$, written $\mathcal{S}\|\mathcal{E}$, is the model under verification, $\mathcal{M}$. The form of the composition depends on the context; we use both synchronous and asynchronous compositions. We will represent the closed system $\mathcal{M}$ as a transition system $(\mathcal{V}_\mathcal{M}, \mathit{Init}_\mathcal{M}, \mathcal{R}_\mathcal{M})$, where the elements respectively denote state variables, initial states, and transition relation. In all of our examples, the environment $\mathcal{E}$ is stateless, generating completely arbitrary inputs to $\mathcal{S}$ at each step; thus $\mathcal{V}_\mathcal{M} = \mathcal{V}$, $\mathit{Init}_\mathcal{M} = \mathit{Init}$ and $\mathcal{R}_\mathcal{M} = \mathcal{R}$.

This paper is concerned with verification of temporal safety properties of the form $\mathbf{G}\Phi$, where $\mathbf{G}$ is the temporal operator "always" and Φ is a state invariant of the form

$$\forall x_1, \ldots, x_k.\ \phi(x_1, \ldots, x_k) \tag{1}$$

where ϕ is a Boolean expression following the syntax of *bE*. The parameters $x_1, \ldots, x_k$ are bit-vector valued, but usually too large to exhaustively case split on.

Example 2: In our running example, we verify $\mathbf{G}\Phi_2$, where

$$\begin{aligned}\Phi_2 \doteq\ & \forall x.\ (\mathit{addr} = x) \rightarrow \\ & \quad ((\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0) \rightarrow \\ & \quad\quad \mathit{cache.data} = \mathit{mem}[\mathit{addr}])\end{aligned} \tag{2}$$

The problem tackled by this paper, temporal safety verification for systems with large data structures, is formally defined as follows.

Definition 1 (Large Data Safety Verification): Given a model $\mathcal{M}$ formed as a composition of system $\mathcal{S}$ and its environment $\mathcal{E}$, and a temporal safety property $\mathbf{G}\Phi$, determine whether or not $\mathcal{M}$ satisfies $\mathbf{G}\Phi$. $\Box$

This problem is known to be undecidable in general since a two-counter machine can be encoded in our formalism using applications of uninterpreted functions [16]. Hence, we can only devise a semi-decision procedure for the problem. In the next section, we describe such a procedure that is based on abstraction.

IV. Methodology

S^{2}W is based on a combination of abstraction and Bounded Model-Checking (BMC). We tackle state-space explosion by abstracting away all but a small subset of the space of the system. We call this mostly-abstracted system our "small world." The abstracted portion of the system can be considered as being updated with an arbitrary value ("$*$") at each step of execution. All other parts of the system are modeled precisely. Thus, this abstraction is a form of localization abstraction [17], where the localization is to small, finite portions of large data structures.

We check the safety property on the small world using BMC. To make BMC sound, we first find and prove the length of the diameter D of our small world to use as the bound – i.e., D is an integer such that every state reachable in $D+1$ steps is also reachable in D or fewer steps. Proving that a conjectured diameter D is correct is undecidable in our formalism [10]. The key to our approach is a set of heuristics that are effective in our chosen application domain of emulators and hypervisors. For our examples, the diameter of the mostly-abstracted system is typically small; we therefore term this the "short world."

If BMC runs for D steps and does not find a violation of the safety property in our small world, then the original model is safe. If BMC finds a counter-example, we cannot say whether the property holds for the original model: BMC can return a spurious counter-example. Choosing the small world well reduces the likelihood of finding spurious counter-examples.

To summarize, there are two crucial pieces to our approach: choosing the right small world and proving the length of the short world. We discuss both of these in more detail below.

As an optimization, we prefix the above approach with an attempt to prove the safety property using one-step induction (on the original, non-abstract model, $\mathcal{M}$). If that succeeds, there is no need to continue on to S^{2}W's abstraction. (This step can be generalized to perform k-step induction as needed.)

For the presentation in this section, it is convenient to represent the system under verification $\mathcal{S}$ as a transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, \mathit{Init})$ where the elements of the tuple have the same meanings as in Sec. III. The environment $\mathcal{E}$ sets the values of the input variables in $\mathcal{I}$ at each step; in all our case studies, the inputs from $\mathcal{E}$ are completely unconstrained. Verification (using induction or BMC) is performed on the composition of $\mathcal{S}$ and $\mathcal{E}$.

A. Induction

First, S^{2}W attempts to prove the safety property using simple one-step induction on the non-abstract model $\mathcal{M}$. We check the validity of the following two formulas, as per standard practice:

$$\mathit{Init}_\mathcal{M}(\mathcal{V}_\mathcal{M}) \rightarrow \Phi(\mathcal{V}_\mathcal{M}) \tag{3}$$

$$\Phi(\mathcal{V}_\mathcal{M}) \wedge \mathcal{R}_\mathcal{M}(\mathcal{V}_\mathcal{M}, \mathcal{V}'_\mathcal{M}) \rightarrow \Phi(\mathcal{V}'_\mathcal{M}) \tag{4}$$

If both checks pass, the verification is complete. We report "Property valid" and exit. If check (3) fails, the property is invalid. We report "Property invalid in initial state" and exit. If check (4) fails, we continue with S^{2}W, to find the small world.

B. Small World

The objective of this step is to identify a small portion of system state that we should model precisely during BMC. Everything else will be allowed to take on arbitrary values at each step of execution.

It is important to note that the soundness of S^{2}W does *not* depend on the choice we make for the small world; we could randomly select some portion of the state to model precisely, abstract everything else away, and if our three steps complete and verify the property, the property would be true of the original, non-abstracted system. However, choosing the small world wisely ensures that the short world is indeed short, which allows BMC to complete in a reasonable amount of time. A well-chosen small world also reduces the amount of spurious counter-examples returned by the BMC step.

We present here a heuristic for choosing the small world when dealing with systems involving large or unbounded data structures. In our case studies, the heuristic found a small world whose short world was reasonable in length and for which no spurious counter-examples were returned by the BMC.

To select those state variables to model precisely, S^{2}W starts with the property $\mathbf{G}\ \Phi$, where Φ is of the form $\forall x_1, x_2, \ldots, x_n.\ \phi(x_1, x_2, \ldots, x_n)$. If we prove Φ by instantiating the quantifier with a completely arbitrary, symbolic parameter vector $(a_1, a_2, \ldots, a_n)$, that suffices to prove the original property. Thus, starting with the symbolic vector $(a_1, a_2, \ldots, a_n)$, we compute a *dependence set* ($\mathcal{U}$) for the instantiated property $\phi(a_1, a_2, \ldots, a_n)$. $\mathcal{U}$ is a *set of expressions* involving state variables and the parameters $a_1, \ldots, a_n$ such that fixing the values of the expressions in this set fixes the value of the instantiated property. For variable M modeling a memory, these expressions typically involve indexing into M at a finite number of (symbolic) addresses. For a Boolean or bit-vector variable, either the variable is in $\mathcal{U}$ or not.

Typically, this set of expressions is derived syntactically by traversing the expression graph of the formula ϕ represented in terms of state and input variables (after performing certain simplifications).

Example 3: In our running example, recall that the property is $\mathbf{G}\ \Phi_2$ where:

$$\begin{aligned}\Phi_2 \doteq\ & \forall x.\ (\mathit{addr} = x) \rightarrow \\ & \quad ((\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0) \rightarrow \\ & \quad\quad \mathit{cache.data} = \mathit{mem}[\mathit{addr}])\end{aligned}$$

Φ_2 has the form $\forall x.\ \phi(x)$. Instantiating x with a, a fresh symbolic constant, we can drop the quantifer and get $\phi(a)$, for which, by propagating the equality $\mathit{addr} = a$, we see that its value is determined by the expressions $\mathit{mem}[a]$ and cache. Thus, we use $\mathcal{U} = \{\mathit{mem}[a], \mathit{cache}\}$ as our dependence set.

Once we have computed $\mathcal{U}$, using the above heuristic or some other method, we can define our small world. Recall that $\mathcal{S}$ is represented as a symbolic transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, \mathit{Init})$. Let $\hat{\mathcal{R}}$ be a transition relation that differs from $\mathcal{R}$ by setting all state not in $\mathcal{U}$ to a non-deterministic value and leaving all others unchanged. Abusing notation slightly to use $\mathcal{U}$ wherever we use $\mathcal{V}$, this means that $\hat{\mathcal{R}}(\mathcal{U}, \mathcal{I}, \mathcal{U}') = \mathcal{R}(\mathcal{U}, \mathcal{I}, \mathcal{U}')$, and $\hat{\mathcal{R}}(\mathcal{W}, \mathcal{I}, \mathcal{W}') = \mathbf{true}$ for $\mathcal{W} = \mathcal{V} \setminus \mathcal{U}$. Similarly, $\hat{\mathit{Init}}(\mathcal{U}) = \mathit{Init}(\mathcal{U})$ and $\hat{\mathit{Init}}(\mathcal{W}) = \mathbf{true}$.

Then the abstracted small world is $\hat{\mathcal{S}} = (\mathcal{I}, \mathcal{V}, \hat{\mathcal{R}}, \hat{\mathit{Init}})$. $\hat{\mathcal{S}}$ is an overapproximate (abstract) version of $\mathcal{S}$ that precisely tracks only the state in $\mathcal{U}$, and allows all other variables to change arbitrarily at each step of execution. Thus, the composition of $\hat{\mathcal{S}}$ and $\mathcal{E}$ is an overapproximate model $\hat{\mathcal{M}}$. (Note that if $\mathcal{M}$ was infinite-state, $\hat{\mathcal{M}}$ is too.)

The next step is proving a short world for $\hat{\mathcal{M}}$ and using BMC on $\hat{\mathcal{M}}$ to verify the property.

C. Short World

The objective of this phase is to determine a bound on the diameter D of the abstract model $\hat{\mathcal{M}}$. For this section, we will assume that $\mathcal{E}$ is stateless, as is the case for all of our case studies; the approach extends in a straightforward manner for the general case. Thus, the diameter of $\hat{\mathcal{M}}$ is the same as that of $\hat{\mathcal{S}}$.

Suppose we believe the diameter to be $\leq k$. To verify this bound, we check the validity of the following logical formula:

$$\begin{aligned}&\forall \mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}.\\ &\quad \big[\mathit{Init}(\mathcal{V}_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{V}_i, \mathcal{I}_{i+1}, \mathcal{V}_{i+1})\big]\\ &\rightarrow \big[\exists \mathcal{V}'_0, \mathcal{V}'_1, \ldots, \mathcal{V}'_k, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_k.\\ &\quad \mathit{Init}(\mathcal{V}'_0) \wedge \bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{V}'_i, \mathcal{I}'_{i+1}, \mathcal{V}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{V}_{k+1} = \mathcal{V}'_i\big] \quad (5)\end{aligned}$$

Since $\hat{\mathcal{R}}$ modifies state expressions outside $\mathcal{U}$ arbitrarily on each step, we can replace $\mathcal{V}$ everywhere in the above formula with $\mathcal{U}$, and obtain the actual convergence criterion that must be checked.

Nevertheless, checking the convergence criterion is undecidable for the class of systems we are interested in, due to the presence of uninterpreted functions, memories, and possibly parameters with unbounded bitwidth [10]. The quantified formula in (5) is also very hard to solve in practice. Therefore, quantifier instantiation heuristics must be devised to perform the convergence check. In this section, we present two such heuristics that have worked well for the range of case studies considered in this paper.

The Sub-Sequence Heuristic: The first heuristic checks that for any state reachable in $k+1$ steps using $k+1$ symbolic inputs to $\hat{\mathcal{S}}$, one can also reach that state using *some sub-sequence of length* $\leq k$ of those $k+1$ symbolic inputs. We can express the sub-sequence heuristic as performing a particular instantiation of the existential quantifiers in criterion (5), and checking the validity of the following formula that results:

$$\begin{aligned}&\forall \mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}, \mathcal{U}'_0, \mathcal{U}'_1, \ldots, \mathcal{U}'_k.\\ &\quad \big[\mathit{Init}(\mathcal{U}_0) \wedge (\mathcal{U}_0 = \mathcal{U}'_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{U}_i, \mathcal{I}_{i+1}, \mathcal{U}_{i+1})\big] \rightarrow\\ &\quad \bigvee_{(\mathcal{I}'_1, \ldots, \mathcal{I}'_k) \prec (\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})} \left[\bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{U}'_i, \mathcal{I}'_{i+1}, \mathcal{U}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{U}_{k+1} = \mathcal{U}'_i\right]\end{aligned} \quad (6)$$

Here the symbol $\prec$ denotes that $(\mathcal{I}'_1, \ldots, \mathcal{I}'_k)$ is a sub-sequence of $(\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})$.

The intuition for the sub-sequence heuristic is that in many systems with large arrays and tables, locations in those tables are updated destructively based on the current input, meaning that past updates do not matter. The address translation logic in the emulators we have studied has this nature. Thus, for such systems, it is possible to drop from the input sequence inputs that have no effect on the $k+1$-st step.

Observe the quantifier alternation in criterion (5) has been eliminated in the stronger criterion (6). Thus, we can simply perform a validity check of an SMT formula in the combination of theories required by our model. If the sub-sequence criterion (6) holds, then so does (5). However, it is possible that criterion (6) is too strong, even when a short diameter exists. This scenario necessitates an alternative semi-automatic approach, described next.

The Gadget Heuristic: The *gadget heuristic* is an approach to instantiating the existential quantified variables in criterion (5) that is particularly useful for systems in which some state in $\mathcal{U}$ depends on the past history of state updates in a non-trivial manner. A gadget is a small sequence of state transitions manually constructed to generate some subset of all reachable system state. A *universal gadget set* is a set of such sequences that, in concert, can generate any reachable system state[1]. The length k of the longest gadget in the universal gadget set is then an upper bound on the diameter of the system.

In terms of the formula expressed as criterion (5), a gadget is a particular guess for a set of initial states $\mathcal{V}'_0$ (expressed symbolically) and a sequence $\mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l$ (for $l \leq k$) of symbolic input expressions to use. For a finite number of gadgets, the inner existential quantifier in criterion (5) can be replaced as a disjunction over all the formulas obtained by substituting the gadget expressions for $(\mathcal{V}'_0, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l)$. If this instantiated formula is valid, then so is the original formula (5).

We defer further discussion about gadget construction to Sec. IV-D, where we discuss its use on our running example.

Performing BMC: Once we have proven k is an upper bound on the length of the diameter of $\hat{\mathcal{S}}$, we run BMC on $\hat{\mathcal{S}}$ for k steps. If $\phi(a_1, \ldots, a_n)$ holds at each step of the simulation, then it follows that $\hat{\mathcal{S}}$ satisfies $\mathbf{G}\ \Phi$. Because $\hat{\mathcal{S}}$ is an overapproximation of all states reachable by $\mathcal{S}$, it follows that $\mathcal{S}$ satisfies $\mathbf{G}\ \Phi$.

If BMC fails, we return a "short" counter-example. The counter-example will be no longer than k. If this is a valid counter-example, the property does not hold. If it is a spurious counter-example, we can return to step two of $\mathrm{S}^2\mathrm{W}$ and expand our set $\mathcal{U}$ to include more state variables and inputs. Such a strategy would be an instance of counterexample-guided abstraction refinement.

Restricted State Spaces: In some systems, we are interested in proving a safety property over a restricted state space, where the restriction can be captured by a predicate over state variables. The restriction predicate is often specified as an antecedent in the temporal safety property. Examples of such a restriction can be found in Sections V-A and V-B. In such cases, we note that it is enough to compute a bound on the reachability diameter — the short world bound — under that restriction. It also sufficient to perform model checking under this restriction.

D. Example

To illustrate the above approach, we apply it to our example. In step one we attempt to prove property $\mathbf{G}\ \Phi_2$ by induction. For this, we perform the following two checks:

$$Init(mem, cache) \rightarrow \Phi_2 \tag{7}$$

$$\Phi_2 \wedge \mathcal{R}_T(\mathcal{V}, \mathcal{V}') \rightarrow \Phi_2 \tag{8}$$

[1]Our gadgets are inspired by "state-generation gadgets," used for automated testing of CPU emulators from arbitrary but reachable initial states [19], and by gadgets identified for return-oriented programming, used to produce a Turing-complete command set for malicious exploits [26].

Check (7) passes, since the cache is initially empty. However, the induction step (check (8)) does not pass. Starting from a state in which Φ_2 holds, it is possible to transition to a state in which Φ_2 is violated. To see why this is so, consider the following state for the cache and two particular entries of *mem*:

$$mem[i] := a,\ mem[j] := b,\ cache.addr := i,\ cache.data := z$$

where $z \neq a$, the last read was for address j, and the output was b. (This state is not reachable in our model, but one-step induction does not take this into account.) Note that Φ_2 holds in this state: for every $x \neq j$, the antecedent ($addr = x$) of the property is false and therefore the property is true; when $x = j$, the nested antecedent ($cache.addr = addr \wedge cache.addr \neq 0$) is false and therefore the property is true. In this state, a $read(i)$ command will hit in the cache and the output will be z, making the property evaluate to false in the next state.

Since simple induction failed for our toy example, we move to the next step, identifying the small world $\hat{\mathcal{S}}_T$. As described in Section IV-B, we introduce a fresh symbolic constant a for x, removing the $\forall x$ quantifier from the property. We then select $\mathcal{U}$ syntactically from the property to be the set of expressions $\mathcal{U} = \{mem[a], cache\}$. In $\hat{\mathcal{S}}_T$ the variables in $\mathcal{U}$ are updated according to the original model ($\mathcal{S}_T$). All other state variables (all entries of *mem* other than *mem*$[a]$) are made to be fully abstract: they are allowed to update to non-deterministic values on every step. The same symbolic constant is used throughout the following short world checks.

The last step of our verification is to identify a short world and then run BMC on the abstract model for the length of the short world. We describe the gadget heuristic here; the sub-sequence heuristic would also work, although it finds a slightly looser bound on the length of the diameter. To build the gadgets we enumerate the possible end-state valuations for the system's state variables (*cache*, *mem*$[a]$) and for each, determine how to get there from a possible starting state. Notice that we only need to consider *mem*$[a]$ and not all of *mem*. This is because, in our small world $\hat{\mathcal{S}}_T$, all entries of *mem* other than *mem*$[a]$ receive new arbitrary values at the end of each step, so we know they can hold any possible value at every step of any trace. In theory there are 2^{34} end-states: one for each possible value of *cache.addr* times the two possible values of *cache.data* and *mem*$[a]$ each. However, for our property, we do not really care about the precise valuation of *cache.addr*, rather, we care about whether $cache.addr = a$ and whether $cache.addr = 0$. So we can abstract away the details of *cache.addr* and consider the following 16 ending states:

$$\begin{aligned}\{cache.addr = a, cache.addr \neq a\}\\ \times \{cache.addr = 0, cache.addr \neq 0\}\\ \times \{cache.data = 0, cache.data = 1\}\\ \times \{mem[a] = 0, mem[a] = 1\}\end{aligned}$$

Not all of the above 16 states are reachable, and in the end four gadgets are enough to reach all reachable states. Each gadget uses either one or two read commands. We build the gadgets with the appropriate values for *addr* and show they form a universal gadget set and therefore, that the short world

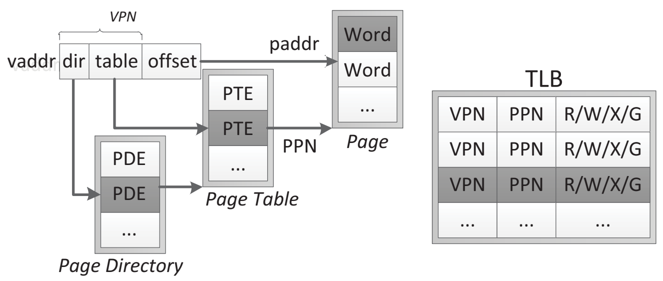


Fig. 3: On the left, we show a two-level page walk translating VPN to PPN addresses. The TLB caches VPN to PPN translations along with read/write/execute/global permission bits.

has length two. We then perform BMC and verify that the property holds.

V. Evaluation

We have evaluated S^2W on six case studies and describe them here: the TLB of the Bochs x86 emulator, a set-associative cache, shadow paging in a hypervisor, hypervisor integrity for SecVisor [12], the Chinese Wall access-control policy in sHype [12], and separation in the original version of ShadowVisor [11]. We describe the first three in detail; the last three were verified using one-step induction and we describe them only briefly. The code for all of our models, along with their verification, is available online.[2] All experiments were performed using UCLID [3] (with the Plingeling SAT solver [1] backend) on a machine with 8 Intel Xeon cores and 4 GB RAM.

A. Bochs' TLB

Bochs [23] is an open source x86 emulator (in C++) for emulating CPU, BIOS, and I/O peripherals. Bochs emulates virtual memory using paging, which includes logic to translate a virtual address (VPN) to a physical address (PPN). Figure 3 illustrates the steps of a page walk. The input virtual address *vaddr* is partitioned into 3 sets of bits $(vaddr_{dir}, vaddr_{table}, vaddr_{offset})$. First, the $vaddr_{dir}$ bits index a page directory entry (PDE) within the page directory region. The PDE contents, along with the $vaddr_{table}$ bits, index into the page tables to retrieve a page table entry (PTE). The PTE contents identify a 4KB physical page, and when concatenated with the 12 bit $vaddr_{offset}$ index a particular byte within this page. Since the above page walk includes two memory lookups, most x86 processors implement a TLB to cache VPN to PPN translations. The TLB also caches permission bits (r/w/x/g) checked during memory accesses. With this optimization, Bochs' address translation logic first checks its TLB for an entry describing the wanted VPN. If no such entry exists, Bochs performs a page walk to compute the corresponding PPN, and then stores that translation in its TLB for future accesses. We would like to prove that the optimized paging unit (with Bochs TLB) is functionally equivalent to the original paging unit (without TLB).

The Bochs TLB + page table system is modeled as a tuple $\mathcal{S}_{Bochs}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, $Init$, $\mathcal{A}$) where

- $\mathcal{I} = \{vaddr, data, pl, rwx, command\}$. *vaddr* is the virtual address to translate. *data* is used to update the page table memory. *pl* indicates the CPU's current privilege level (either user or supervisor mode). *rwx* indicates whether this memory access writes and/or executes this address.
- $\mathcal{O} = \{paddr\_TLB, pagefault\_TLB, paddr\_noTLB, pagefault\_noTLB\}$. *paddr_TLB* is the result of address translation with TLB. *paddr_noTLB* is the result of address translation without the TLB. *pagefault_TLB* indicates a page fault occurred (due to insufficient permission) during translation with TLB. *pagefault_noTLB* indicates a page fault during translation without the TLB.
- $\mathcal{V} = \{mem, TLB, legal\}$, where *mem* is a 32-bit addressable memory containing both the page directory and page tables. *TLB* is an array (2^{10} entries in Bochs) of structs, where each struct is 160 bits wide and has 5 32-bit fields: *vpn*, *ppn*, access bits (*ab*), etc. *legal* is a Boolean variable denoting whether the system reached the current state via a legal sequence of transitions.
- $Init = (\text{mem}_0, TLB_0, \mathbf{true})$, where $TLB_0[i].vpn :=$ 0xffffffff for all i and mem_0 is an uninterpreted function from 32 bit addresses to arbitrary 32 bit values. The Bochs TLB is initialized with its *vpn* field set to 0xfffffff in all entries, thus making it empty. *legal* is initialized to true.
- $\mathcal{A}$: $\mathcal{V}$ evolves via operations *write_pde*, *write_pte*, *invlpg*, *invlpg_all*, *setcr3*, and *translate*, and the environment non-deterministically chooses one of these operations at each step. Table I describes each of these commands.

Command	Modifies	Guard
write_pte	*mem*	**true**
write_pde	*mem*	**true**
translate	*TLB*	$\neg present \vee \neg permission$
set_cr3	*TLB*	**true**
invlpg	*TLB*	$TLB[vaddr_{table}].vpn_{31:12} = (vaddr_{dir} \circ vaddr_{table})$
invlpg_all	*TLB*	$TLB[vaddr_{table}].vpn_{31:22} = vaddr_{dir}$

TABLE I: The allowable operations in our model of the Bochs TLB.

Each command is implemented in distinct functions within Bochs (src/cpu/paging.cc). Since Bochs executes on a single thread, we can safely model each function as an atomic operation, i.e., a single step in the state transition system. The commands *write_pde* and *write_pte* are used to update the page directory and page tables respectively, typically to modify access permissions or page mapping. *translate* performs address translation and assigns the result to variables in $\mathcal{O}$. Furthermore, if a page walk was deemed necessary, then *translate* updates a TLB entry with the results of that page walk. If global pages are enabled, then a *setcr3* (which switches to a new page table, typically during a context switch) flushes all non-global entries in the TLB. Otherwise if global pages are disabled, all TLB entries are flushed on a *setcr3*. The x86 instruction *invlpg* flushes a specific TLB entry containing the translation for *vaddr*; *invlpg* is needed to invalidate the TLB entry following a write to the page table. *invlpg_all* atomically flushes all TLB entries that have $vaddr_{table}$ in their *vpn* (bits 31 to 22); *invlpg_all* is needed to invalidate a set of TLB entries following a write to the page directory.

We check equivalence of both the physical address and whether a page fault occurred. Since the x86 manual only guarantees cache coherency when the TLB is flushed properly, we only require equivalence on traces where each *write_pde*

[2]http://uclid.eecs.berkeley.edu/s2w/

is followed by a *invlpg_all* and each *write_pte* is followed by *invlpg*. This constraint is enforced by *legal*, which is true only if the sequence of operations abides by these constraints. Any state that satisfies *legal* is guaranteed to be reachable from the initial state via a legal sequence of state transitions.

The property that we check is:

$$\Phi_9 \doteq \forall v, p, r.\ (vaddr = v \wedge pl = p \wedge rwx = r) \rightarrow \\ legal \rightarrow ((pagefault\_TLB \Leftrightarrow pagefault\_noTLB) \wedge \\ (\neg pagefault\_noTLB \rightarrow (paddr\_noTLB = paddr\_TLB))) \quad (9)$$

1) Induction: The one-step induction check consists of proving (10) and (11) using the UCLID verifier.

$$Init(\mathcal{V}_{Bochs}) \rightarrow \Phi_9(\mathcal{V}_{Bochs}) \quad (10)$$

$$\Phi_9(\mathcal{V}_{Bochs}) \wedge \mathcal{R}(\mathcal{V}_{Bochs}, \mathcal{V}'_{Bochs}) \rightarrow \Phi_9(\mathcal{V}'_{Bochs}) \quad (11)$$

Our initial state satisfies Φ_9 because the TLB is initialized to be empty, thereby forcing both optimized and unoptimized designs to undergo the two-level page walk. However, one-step induction (check (11)) fails because the back-end SMT engine cannot solve the formula, which has a quantifier alternation. Consequently, we proceed onto the small and short world steps.

2) Small World: We syntactically derive the dependence set $\mathcal{U}_{\Phi_9}$ by traversing the expression graph of Φ_9. After introducing a fresh 32-bit symbolic constant $v = (v_{dir}, v_{table}, v_{offset})$, the dependence set is

$$\mathcal{U}_{\Phi_9} = \{legal, TLB[v_{table}], mem[cr3_{31:12} \circ v_{dir}], \\ mem[mem[cr3_{31:12} \circ v_{dir}]_{31:12} \circ v_{table}]\}$$

The last three expressions represent the TLB entry, page directory entry, and page table entry pointed to by v, respectively. (Here $cr3_{31:12}$ refers to the upper 20 bits of the cr3 control register and is modeled as a symbolic constant.) Our abstract model $\hat{\mathcal{S}}_{Bochs}$ precisely tracks only the variables in $\mathcal{U}_{\Phi_9}$; other state elements get updated with arbitrary values at each step.

3) Short World: We use the sub-sequence heuristic to find an upper bound on the diameter of $\hat{\mathcal{S}}_{Bochs}$. We find a bound of 9 steps. Finally, we perform bounded model checking for 9 steps, proving that all reachable states of $\hat{\mathcal{S}}_{Bochs} \| \mathcal{E}_{Bochs}$ satisfy Φ_9. The sub-sequence check and BMC took about 45 minutes and 25 minutes respectively.

B. Content Addressable Memory

While the TLB functions as a direct-mapped cache (each concrete logical address is associated with a single TLB entry), S^{2}W also applies to systems with set-associative caches and Content Addressable Memories (CAMs). A CAM stores associations between keys and data. The key is typically stored as part of the data, and is used for comparison during lookups.

Figure 4 shows a system containing slow memory and a CAM-based cache. We would like to prove that a lookup in slow memory yields the same data as lookup in the CAM-based cache (if the data is present in the CAM). We model the CAM's state using a variable *cam* that maps a CAM index to its contents, a 65-bit vector containing fields *present*, *key*, and *data*. The $cam[i].present$ bit indicates whether the key $cam[i].key$ and data $cam[i].data$ are valid entries. $cam[i].key$

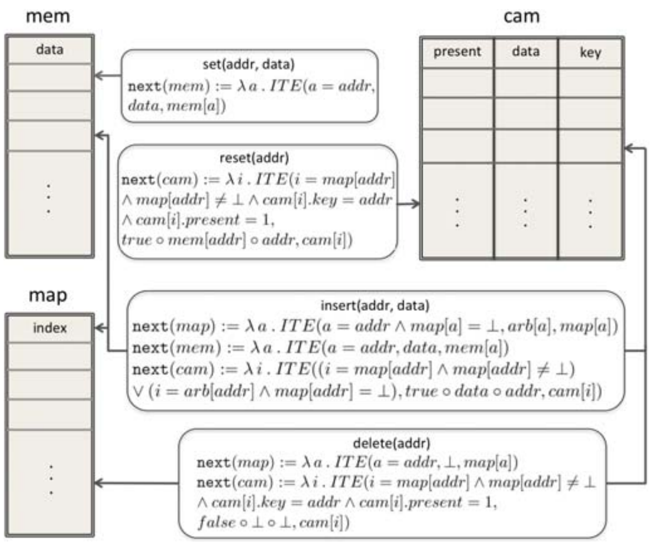

mem
data
cam
present
data
key
set(addr, data)
$\mathtt{next}(mem) := \lambda a\,.\,ITE(a = addr, data, mem[a])$
reset(addr)
$\mathtt{next}(cam) := \lambda i\,.\,ITE(i = map[addr] \wedge map[addr] \neq \bot \wedge cam[i].key = addr \wedge cam[i].present = 1, true \circ mem[addr] \circ addr, cam[i])$
map
index
insert(addr, data)
$\mathtt{next}(map) := \lambda a\,.\,ITE(a = addr \wedge map[a] = \bot, arb[a], map[a])$
$\mathtt{next}(mem) := \lambda a\,.\,ITE(a = addr, data, mem[a])$
$\mathtt{next}(cam) := \lambda i\,.\,ITE((i = map[addr] \wedge map[addr] \neq \bot) \vee (i = arb[addr] \wedge map[addr] = \bot), true \circ data \circ addr, cam[i])$
delete(addr)
$\mathtt{next}(map) := \lambda a\,.\,ITE(a = addr, \bot, map[a])$
$\mathtt{next}(cam) := \lambda i\,.\,ITE(i = map[addr] \wedge map[addr] \neq \bot \wedge cam[i].key = addr \wedge cam[i].present = 1, false \circ \bot \circ \bot, cam[i])$

Fig. 4: Our model of a slow memory and its CAM-based cache.

and $cam[i].data$ contain the 32-bit key and 32-bit data stored at CAM index i (if $cam[i].present$ is true). Memory is modeled as an variable *mem* mapping a 32-bit address to a 32-bit vector. That is, $mem[a]$ refers to the 32-bit data at address a.

We also maintain a state variable *map* that maps an address a to a 32-bit CAM index. *map* is updated when data is added or deleted from the CAM. A $read(addr)$ command checks the contents of the CAM at index $map[addr]$. If $cam[map[addr]].key = addr$ and $cam[map[addr]].present$ is true, then $\mathcal{S}_{CAM}$ assigns $cam[map[addr]].data$ to output variable out_cam_{data} and true to output variable $out\_cam_{present}$. Otherwise, $\mathcal{S}_{CAM}$ assigns false to $out\_cam_{present}$. *read* also assigns $mem[addr]$ to output variable out_mem_{data}. The $insert(addr, data)$ command checks *map* for an existing mapping of address $addr$. If $map[addr] \neq \bot$, then $\mathcal{S}_{CAM}$ updates the CAM at index $map[addr]$ with data *data* and key *addr*.[1] Otherwise, we arbitrarily choose a new location $arb[addr]$ to insert *data* and *addr*, and update $map[addr]$ with this new location. The $set(addr, data)$ command updates $mem[addr]$ with *data*, possibly making the CAM contents stale. The $reset(addr)$ command resynchronizes *cam* with *mem* at address *addr* (if the CAM contains a valid entry for address *addr*). These commands are implemented using atomic operations, and they update *map*, *mem* and *cam* in parallel.

The CAM + memory system is modeled as a tuple $\mathcal{S}_{CAM}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, $Init$, $\mathcal{A}$) where

- $\mathcal{I} = \{addr, data, command\}$
- $\mathcal{O} = \{out\_cam_{data}, out\_cam_{present}, out\_mem_{data}\}$
- $\mathcal{V} = \{cam, mem, map, legal\}$: These are all modeled as bit-vector functions. *cam* returns a 65-bit vector: 1-bit *present*, 32-bit *data*, 32-bit *addr*. Both *mem* and *map* return a 32-bit vector. *legal* is a Boolean variable denoting whether the system legally reached the current state.
- $Init = (c_0, m_0, map_0, \mathbf{true})$, where $map_0[a] = \bot$ for all a, the present field of each CAM entry is initialized to false, memory is initialized to arbitrary values, and *legal* is initialized to true.

[1]Note that $\bot$ in our model equals 0x00000000; it is an acceptable design choice to not cache that address.

- $\mathcal{A}$: The state evolves via commands *insert*, *delete*, *set*, *reset* and *read*. The environment non-deterministically chooses one of these commands at each step. Figure 4 defines the state transition relation for each command.

The safety property Φ_{12} checks that the CAM and memory have the same data for all keys present in CAM. Note that the CAM only guarantees cache coherency if it is resynchronized with memory after each *set*. The state variable *legal* enforces this constraint: it is true if every *set* is followed by a *reset*.

$$\Phi_{12} \doteq \forall a.\ (addr = a) \rightarrow legal \rightarrow \\ (out\_cam_{present} \rightarrow (out\_cam_{data} = out\_mem_{data})) \qquad (12)$$

Since Φ_{12} is expressed over output variables, we check if a state s satisfies Φ_{12} by performing a *read* operation on state s with a fresh symbolic constant for a.

1) Induction: We first try one-step induction on this system to prove Φ_{12}. The initial state satisfies Φ_{12} because both the CAM and *map* are empty. However, the inductive check fails because of the quantifier alternation, similar to the TLB case study. Hence, we continue onto the small world step of our approach.

2) Small World: We syntactically derive the dependence set by traversing the expression graph of Φ_{12}. We introduce a fresh 32-bit symbolic constant a for the address that we precisely track in *map* and *mem*. We precisely track *legal*, $map[a]$, $mem[a]$, and $cam[map[a]]$.

$$\mathcal{U}_{\Phi_{12}} = \{legal, cam[map[a]], mem[a], map[a]\} \qquad (13)$$

Our abstract model $\hat{\mathcal{S}}_{CAM}$ precisely tracks updates to only these variables.

3) Short World: Using the sub-sequence heuristic, we find an upper bound on the reachability diameter of 5 steps. Finally, we perform bounded model checking for 5 steps, proving that all reachable states of $\hat{\mathcal{S}}_{CAM} \| \mathcal{E}_{CAM}$ satisfy Φ_{12}. The sub-sequence check and BMC takes about 15 seconds and 5 seconds respectively.

C. Shadow Paging

For our third case study, we model a shadow page table system. A hypervisor may use shadow page tables to assure address space separation between the guest and host. The guest page tables can be updated arbitrarily by the guest operating system, while the shadow page tables are updated only by the hypervisor. The hardware uses the shadow page tables for address translation, so it is the hypervisor's responsibility to make sure the shadow page tables stay synchronized with the guest tables, while at the same time ensuring no translation will ever allows the guest to access memory outside its allocated sandbox. We model the synchronization process and verify that the physical address returned by translation never exceeds some constant limit, LIMIT.

Our shadow paging model (Figure 5) is as follows. There are two page table structures: guest and host. Each is a two-level structure: a page directory table (PDT) and a page table (PT). We refer to the guest and shadow page tables as gPDT, gPT and sPDT, sPT, respectively. Entries in the PDT have three fields: present (p), page-size-extension (pse), and address

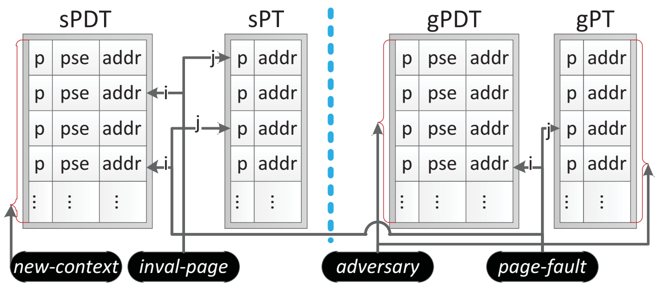


Fig. 5: An illustration of the shadow page table model.

($addr$). Entries in each nested PT have two fields: present (p) and address ($addr$).

Let $\mathcal{S}_{SP}$ = ($\mathcal{I}$, $\mathcal{V}$, *Init*, $\mathcal{A}$) be the shadow paging model with

- $\mathcal{I} = \{i, j, command\}$: i and j index into the PDT and PT, respectively; *command* is one of *page-fault*, *inval-page*, *new-context*, or *adversary*.
- $\mathcal{V} = \{$gPDT, gPT, sPDT, sPT, LIMIT$\}$. gPDT, gPT, sPDT, and sPT are modeled as functions that map indices to bit-vectors. gPDT and sPDT return 34-bit vectors: (1-bit p, 1-bit pse, 32-bit $addr$). gPT and sPT return 33-bit vectors: (1-bit p, 32-bit $addr$). LIMIT is a constant 32-bit vector.
- $Init = (\text{sPDT}_0, \text{sPT}_0, \text{gPDT}_0, \text{gPT}_0)$, where sPDT and sPT are both initialized with the p bit cleared in all entries. gPDT and gPT are initialized to arbitrary values.
- $\mathcal{A}$: The four commands update state in the following way: *page-fault* synchronizes the shadow tables with the guest tables. *inval-page* conditionally invalidates (zeros out) entries in sPDT and sPT. *new-context* unconditionally invalidates entries in sPDT. *adversary* writes to gPDT and gPT. The assignments to gPDT, gPT, sPDT, and sPT are summarized in Table II.

Command	Modifies	Guard
page-fault(i, j)	sPDT$[i]$	gPDT$[i].pse \ \wedge$ gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT)
	sPDT$[i]$	gPDT$[i].pse \ \wedge \neg$(gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT))
	sPDT$[i]$, sPT$[j]$	$\neg$gPDT$[i].pse \ \wedge$ gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT)
	sPDT$[i]$	$\neg$gPDT$[i].pse \ \wedge$ gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ $\neg$(gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT))$\wedge$ $\neg$(sPDT$[i].p \wedge \neg$sPDT$[i].pse$)
	sPDT$[i]$	$\neg$gPDT$[i].pse \ \wedge \neg$(gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT))
	sPT$[j]$	$\neg$gPDT$[i].pse \ \wedge$ gPDT$[i].p \wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ $\neg$(gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT))$\wedge$ (sPDT$[i].p \wedge \neg$sPDT$[i].pse$
inval-page(i, j)	sPDT$[i]$	(sPDT$[i].p \wedge \neg$gPDT$[i].p$)$\vee$ (sPDT$[i].p \wedge$ gPDT$[i].p \wedge$ (sPDT$[i].pse \vee$ gPDT$[i].pse$))
	sPT$[j]$	sPDT$[i].p \wedge$ gPDT$[i].p \wedge$ $\neg$gPDT$[i].pse \wedge \neg$sPDT$(i).pse$
new-context	gPDT	**true**
adversary	gPDT gPT	**true** **true**

TABLE II: Next-state assignments for the shadow paging model.

Our model is based on the ShadowVisor model [11], but has been extended to introduce pointers. The safety properties we verify are similar to those of ShadowVisor. We verify that a translation using the shadow page tables will never return an address above a fixed limit.

$$\Phi_{14} = \forall i.\ (\text{sPDT}[i].p \wedge \text{sPDT}[i].pse) \rightarrow \\ \text{sPDT}[i].addr < \text{LIMIT} \qquad (14)$$

$$\Phi_{15} = \forall i, j.\ (\text{sPDT}[i].p \wedge \neg \text{sPDT}[i].pse \wedge \text{sPT}[j].p) \rightarrow \text{sPT}[j].addr < \text{LIMIT} \quad (15)$$

1) Induction: The property $\mathbf{G}\ \Phi_{14}$ is proven by induction. However, one-step induction fails to prove $\mathbf{G}\ \Phi_{15}$. If sPT$[j]$ is marked as present, but has an address greater than LIMIT, Φ_{15} is still true as long as sPDT$[i]$ is marked not present. From that state, it is possible to update sPDT$[i]$ to present without updating sPT$[j]$, so that in the next state the sPDT$[i]$ and sPT$[j]$ entry are both marked present and the data in the sPT$[j]$ entry is greater than LIMIT, violating Φ_{15}. Therefore, we move on to the small and short world steps for $\mathbf{G}\ \Phi_{15}$.

2) Small World: The properties are concerned with a single entry i in sPDT and a single entry j in sPT. Therefore we use a fresh symbolic constant to choose an arbitrary entry from each. In particular, our conditional dependency set is $\mathcal{U}_{\phi_{15}} = \{\text{sPDT}[a_i], \text{sPT}[a_j]\}$. In our abstract symbolic transition system, $\hat{\mathcal{S}}_{SP}$, we track precisely only the state in $\mathcal{U}_{\phi_{15}}$. a_i, a_j stay constant.

3) Short World: The sub-sequence short-world heuristic does not work for page tables, because page-table entry updates can depend on previous writes to the entry or to other entries; it is not always possible to drop one step of a trace to achieve an equivalent final state. Instead, we use gadgets to find and prove the short world. We manually construct a universal gadget set to prove the length of the diameter of $\hat{\mathcal{S}}_{SP}$.

To build the gadgets we case split on the possible end-state valuations for the variables in $\mathcal{U}_{\phi_{15}}$, and for each, determine how to get there from a valid starting state. The model has only four commands and it was usually obvious which commands were needed to get to a particular state. The gadgets must also specify the parameters (i, j) to the command, and, in the case of the *adversary* command, the value that gets written to the guest tables. Figuring out the correct parameters to use for each command was more difficult. In this case, the parameters were always either $i := a_i$ or $i := a'_i$ (where $a'_i \neq a_i$ is arbitrarily chosen), and similarly for j. The *adversary* data that gets written to the guest tables was a combination of the *addr* field of the (symbolic) end-state valuation we were trying to achieve and a particular value for the *p* and *pse* bits, which we chose according to the particular gadget we were building.

We needed a total of thirteen gadgets, each four commands or less, to prove the short world has length four. We then ran BMC on $\hat{\mathcal{S}}_{SP}$ for four steps and verified the property held at each step. The verification of the short world took less than a minute; BMC took approximately five seconds.

D. Other Hypervisor Models

We applied our abstraction technique to three additional hypervisor models. All were verified using one-step induction, each within 5 seconds.

SecVisor: SecVisor [12], [25] is a small hypervisor that supports a single guest OS. It virtualizes the memory management unit by implementing shadow page tables and synchronizing them with the guest page tables. The model assumes an adversary can write arbitrary values to the guest page tables. SecVisor aims to execute only approved code in kernel mode; therefore, the page-table synchronization must prevent adversary-provided code from having execute permissions while in kernel mode. We verified the security property using one-step induction on a model given by Franklin et al. [12].

sHype: sHype [12] is an access control system used by the Xen [2] hypervisor. Based on the Chinese Wall policy, it establishes "conflict of interest" classes and guarantees each virtual machine will never access two pieces of data from the same conflict of interest class. We verified the security property using a model presented by Franklin et al. [12].

ShadowVisor: ShadowVisor [11] served as the starting point for our shadow page table model. It models the page tables of a simple hypervisor that assures address space separation between guest and host by maintaining separate guest and shadow page tables. Like our shadow page table model, ShadowVisor guarantees that if an address is marked as present, it will never exceed a certain fixed limit. We model ShadowVisor in our modeling language and use one-step induction to verify the property.

VI. Related Work

Verification of infinite-state or parametrized systems has been well-studied. Here we present the most closely-related work.

Franklin et al. [11], [12] present a small-model approach to verifying systems with parametrized data structures (arrays). In essence, they present a formal language such that if the system can be modeled in their language, then a small-model theorem applies, stating that the unbounded arrays can be reduced to arrays with one designated element alone. Finite-state model checking can then be employed on the resulting system. While this approach is very elegant, there are some important differences with the approach in this paper. First, our modeling language is more expressive, allowing us to model the Bochs TLB, CAM, and shadow paging examples which cannot be modeled in their language. Second, our approach is different: we compute an abstraction based on localization within the large data structures, and we use bounded model checking.

The use of inductive invariant checking is common in this problem domain. The method of *invisible invariants* [6], [22] is an example of an inductive verification technique applied to systems of N identical finite-state processes. The core idea in this method is to generalize from the reachable states of a small number of processes into a quantified inductive assertion of the form $\forall i.\phi(i)$, where the index i ranges over process IDs. Namjoshi [24] also discusses the so-called *cutoff method*, which is a small-model approach for such systems of parametrized processes, drawing connections between inductive methods, small-model approaches and compositional reasoning. In general, these approaches are not easily applied to our examples since they are not naturally decomposed into a system of N identical finite-state processes. Instead, in our problem domain, the number of interacting processes is usually finite and small, but the shared data structures are large and complicated.

Abstraction-based approaches have also been presented for infinite-state or parametrized systems similar to those studied in this paper. Lahiri and Bryant [18] presented an approach for verifying universally-quantified invariants on parametrized systems using predicate abstraction. While predicate abstraction can be quite effective for verifying control-related properties, especially when one can guess suitable predicates, it is not suitable for verifying equivalence of two code versions (such as in the Bochs TLB case study), which is a highly data-dependent property. McMillan [21] presents a semi-automatic approach to compositional reasoning using an abstraction similar to ours. Given a system with large arrays, he uses a form of localization abstraction (guided by case splitting performed by the user) to only model a few entries in the arrays precisely, allowing all other entries to be updated by an arbitrary value $\perp$. This abstraction is used to compute a finite-state abstract model on which reachability analysis is performed. In contrast, our method computes an abstraction based on index terms derived from the property, and uses a BMC-based approach to verify the system. Bjesse [8] describes an automatic approach for verifying sequential circuits with large memories, if the memories are "remodellable" (a notion made formal in [8]). The work takes a "small-world" approach, transforming an initial netlist into another with memories with precise updates only to a small number of entries in memories, and uses counterexample-guided abstraction-refinement. Our work does not require the "remodellable" restriction. German [14] presents a novel approach for constructing sound and complete abstractions for similar systems. The approach is based on performing static analysis on the system model, and (in contrast to [8]) can handle unbounded delays between the time the array is read and when the read value propagates to the output. While the approach is completely automatic, it cannot handle certain data structures such as CAMs, in which a read, in principle, requires scanning the entire array. Our approach, while not always automatic, does handle structures such as CAMs (see Sec. V-B).

Efficient memory modeling is a technique used for bounded verification problems, either symbolic simulation (e.g. [27]) or bounded model checking (e.g. [13]). In contrast, our approach focuses on unbounded verification based on abstraction and a sound application of BMC based on heuristics to find the reachability diameter.

There has also been prior work on verifying emulators and hypervisors. Alkassar et al. [5] presented the verification of the TLB logic in the Hyper-V hypervisor. They verify invariant properties of the TLB using the VCC verifier. Their approach, like ours, is not fully automatic. While our approach uses abstraction-based model checking, assuming a particular model of atomicity of operations, theirs is a classic deductive verifier for C code (using VC generation and theorem proving) that operates at a somewhat lower level of abstraction.

VII. Conclusion

We have presented S^{2}W, a new approach to verifying systems with large or unbounded data structures that combines induction and abstraction-based model checking. Experimental results have been presented on several examples of emulators and hypervisors. In ongoing work, we are investigating how to make the technique more automated, to automatically generate abstract, term-level models from C/C++ code, and to validate these models.

Acknowledgments. This research was supported in part by SRC contract 2045.001, by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-09-1-0539, and by a generous gift from Intel. We thank Randal E. Bryant, Orna Kupferman, and Anupam Datta for their valuable feedback.

References

[1] Plingeling SAT Solver. http://fmv.jku.at/lingeling.
[2] The Xen Hypervisor. http://www.xen.org/.
[3] UCLID Verification System. Available at http://uclid.eecs.berkeley.edu.
[4] VMware Security Advisory vmsa-2009-0015. http://www.vmware.com/security/advisories/VMSA-2009-0015.html, 2009.
[5] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. J. Paul. Verifying shadow page table algorithms. In *FMCAD*, 2010.
[6] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *CAV*, 2001.
[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.
[8] P. Bjesse. Word-Level Sequential Memory Abstraction for Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.
[9] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, 2002.
[10] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.
[11] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan. Parametric Verification of Address Space Separation. In *POST*, 2012.
[12] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size. In *IEEE Security & Privacy*, 2010.
[13] M. K. Ganai, A. Gupta, and P. Ashar. Efficient Modeling of Embedded Memories in Bounded Model Checking. In *Proc. Computer-Aided Verification (CAV)*, 2004.
[14] S. German. A theory of abstraction for arrays. In *FMCAD*, 2011.
[15] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *EuroSys*, 2006.
[16] A. J. Isles, R. Hojati, and R. K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In *Computer-Aided Verification (CAV '98)*, 1998.
[17] R. Kurshan. Automata-Theoretic Verification of Coordinating Processes. In *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, volume 199. Springer Berlin / Heidelberg, 1994.
[18] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.
[19] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *ASPLOS*, 2012.
[20] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *ISSTA*, 2009.
[21] K. L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 1999.
[22] K. L. McMillan and L. D. Zuck. Invisible Invariants and Abstract Interpretation. In *SAS*, 2011.
[23] D. Mihocka and S. Shwartsman. Virtualization Without Direct Execution or Jitting : Designing a Portable Virtual Machine Infrastructure. *AMASBT*, 2008.
[24] K. S. Namjoshi. Symmetry and Completeness in the Analysis of Parameterized Systems. In *VMCAI*, 2007.
[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.
[26] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc Without Function Calls (on the x86). In *ACM CCS*, 2007.
[27] M. N. Velev, R. E. Bryant, and A. Jain. Efficient Modeling of Memory Arrays in Symbolic Simulation. In *Proc. Computer-Aided Verification (CAV)*, 1997.
[28] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *VEE*, 2010.

Decompilation into Logic — Improved

Magnus O. Myreen, Michael J. C. Gordon
Computer Laboratory, University of Cambridge, UK

Konrad Slind
Rockwell Collins, USA

***Abstract*—This paper presents improvements to a technique which aids verification of machine-code programs. This technique, called decompilation into logic, allows the verifier to only deal with tractable extracted models of the machine code rather than the concrete code itself. Our improvements make decompilation simpler, faster and more generally applicable. In particular, the new technique allows the verifier to avoid tedious reasoning directly in the underlying machine-code Hoare logic or the model of the instruction set architecture. The method described in this paper has been implemented in the HOL4 theorem prover.**

I. Introduction

Verification of machine-code programs is hopelessly tedious without good tool support, particularly if verification is to be done against realistic models of commercial machine languages, e.g. x86, ARM, PowerPC, MIPS, whose formal models are several thousand lines long. Done naively, verification efforts fail to scale, get tied to a specific architecture model and may require reading or even annotating machine-code programs.

In previous work [13], we have proposed a technique that can significantly ease verification of machine-code programs, namely: decompilation into logic. Given some concrete machine code and a model of an instruction set architecture (ISA), this decompilation extracts functions (defined in logic) which capture the functional behaviour of the machine code. We have demonstrated that this decompilation technique can be used for post hoc verification, as described below, and also for implementation of proof-producing synthesis tools, as described in [14]. We have shown that these techniques scale to significant examples, including verification of functional correctness of garbage collectors and Lisp implementations in ARM, x86 and PowerPC [10], [11], and decompilation of the seL4 microkernel [12] (12,000 lines of ARM).

This paper's contribution is a presentation of significant technical improvements to our decompilation method [13]. The improvements make the new approach faster, simpler and more widely applicable. In particular, the new technique allows the verifier to avoid reasoning directly in the underlying machine-code Hoare logic, even in the presence of code pointers. The new approach retains all of the features of the previous approach and adds a few new ones (Section II-A). The technique described in this paper has been implemented (www.cl.cam.ac.uk/~mom22/decompilation) in the HOL4 theorem prover [16] and applied to ARM machine code.

This document is an overview of MOD sponsored research and is released to inform projects that include high integrity software or complex electronics. The information contained in this document should not be interpreted as representing the views of the MOD, nor should it be assumed that it reflects any current or future MOD policy. The information cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

A. Example: Sum of An Array

We start with an example which illustrates what we mean by decompilation and verification via decompilation. Consider the following C code which calculates the sum of an array. (It ignores indexed 0 to make the ARM assembly neater.)

```
do { k += a[i] } while (--i != 0);
```

The C code above can be compiled to ARM assembly:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

which can be assembled into ARM machine code:

```
E7921003 E0800001 E2533004 1AFFFFFB
```

Decompilation takes concrete machine code as input. From this machine code it extracts a function which describes the behaviour of the code. In this case, sum below which records how registers r0–r3 and memory are affected and also what side condition must hold for correct execution. The side conditions are collected by the *cond* component.

$$
\begin{array}{l}
\mathsf{sum}(cond, r_0, r_1, r_2, r_3, m) = \\
\quad \mathsf{let}\ cond = cond \wedge \mathsf{valid\_address}\ (r_2 + r_3)\ m\ \mathsf{in} \\
\quad \mathsf{let}\ r_1 = m(r_2 + r_3)\ \mathsf{in} \\
\quad \mathsf{let}\ r_0 = r_0 + r_1\ \mathsf{in} \\
\quad \mathsf{let}\ r_3 = r_3 - 4\ \mathsf{in} \\
\quad\quad \mathsf{if}\ r_3 = 0\ \mathsf{then}\ (cond, r_0, r_1, r_2, r_3, m) \\
\quad\quad\quad\quad \mathsf{else}\ \mathsf{sum}(cond, r_0, r_1, r_2, r_3, m)
\end{array}
$$

Decompilation also automatically proves a theorem, which we call a *certificate theorem*, relating the new function to the machine code from which it was extracted. The certificate theorem is derived w.r.t. a model of the ISA of the machine code; in this case, a model of ARM developed by Fox [4]. We state these certificate theorems using a machine-code Hoare triple which is parametrised by the ISA model. The Hoare triples will be explained in later sections, for now read the following certificate theorem for sum informally: for any input $(c, r_0, r_1, r_2, r_3, m)$ which sum relates to output $(c', r'_0, r'_1, r'_2, r'_3, m')$, the execution of the ARM machine code can perform the state update corresponding to sum:

$$
\begin{array}{l}
(\mathsf{sum}(c, r_0, r_1, r_2, r_3, m) = (\mathsf{true}, r'_0, r'_1, r'_2, r'_3, m')) \Longrightarrow \\
\{\ \text{ARM state holds}\ (r_0, r_1, r_2, r_3, m)\ \} \\
\texttt{E7921003 E0800001 E2533004 1AFFFFFB} \\
\{\ \text{ARM state holds}\ (r'_0, r'_1, r'_2, r'_3, m')\ \}
\end{array}
$$

The benefit of using decompilation in verification is that once the machine code has been decompiled, subsequent verification can concentrate on only proving properties of the extracted function, since any property proved of the extracted function applies directly to the machine code via the certificate theorem. For example, with an appropriate definition of how arrays are stored in memory, it is easy to prove that sum correctly sums the content of an array and using the certificate theorem relate this property to the machine code.

II. Improvements

As mentioned above, this paper's contribution is to present improvements to the decompilation approach. In particular, we show how decompilation into logic can be made simpler, faster and more generally applicable.

Simpler: The original approach to decompilation was geared towards automating proofs in a Hoare logic that was intended for manual proofs — a complicated Hoare logic that was never optimised for mechanisation performance. In this paper, we show that a much simpler Hoare logic can be used for decompilation. The new Hoare logic (Section III-A) is only a thin layer over the model of the ISA.

Faster: In the new approach, we carefully state the intermediate theorems so that composition of intermediate results can be done in a handful of fast operations. In the previous approach, composition was the main performance bottleneck: often involving simplification through rewriting and calculation of a separation logic 'frame'. The new approach to composition is described in Section III-B. The speed-up we gained can seen in benchmarks listed in Figure 1.

More widely applicable: The main practical drawback of the previous approach was its inability to deal with code involving exotic control flow (e.g. code using more than just goto-like jumps). This lead to an unsatisfactory compromise where certain complicated code had to be verified manually using the machine-code Hoare triples. The new approach is engineered so that it successfully extracts a function even in the presence of code pointers. With the new approach, one can practically always avoid reasoning directly in the underlying Hoare logic. The new approach extracts a single function from the given machine code as before if possible; otherwise, it extracts a function which describes each chunk of well-behaved code. The example below will illustrate what we mean.

A. Example: Calling Every Code Pointer of An Array

To illustrate how the new decompilation approach deals with complicated control-flow, consider the following example program which calls each code pointer stored in an array.

```
  do { (a[i])() } while (--i != 0);
```

The C code above can be compiled to ARM assembly:

```
L0: ldr r4,[r5,r6]    ; load mem[r5+r6] into r4
L1: blx r4            ; call code-pointer r4
L2: subs r6,#4        ; decrement r6 by 4
L3: bne L0            ; goto L0 if r6 ≠ 0
L4:
```

Our previous approach to decompilation is not able to process the resulting ARM code, because it is unclear what function describes the effect of the call to the code pointer. In the new approach, we avoid this issue by essentially leaving 'holes' in the extracted function.

The ARM code above decompiles into a function which explicitly mentions the value of the program counter pc. The extracted function has three parts; the first part describes the effect of starting execution from the top of the code ($pc = \texttt{L0}$): in this case, a load is performed and a call is made to a code pointer, i.e. the pc is updated (with a word-aligned address) and a return address is stored in r_{14}. Note that this function does not make any assumption that the call to the code pointer returns (it might not). The second part of the function describes what happens if execution returns ($pc = \texttt{L2}$): in this case r_6 is decremented and control moves either to the top or bottom of the code. The third case just states that all other cases are ignored, i.e. no progress is made.

```
calls$(cond, pc, r_4, r_5, r_6, r_{14}, m)$ =
  if $pc = \texttt{L0}$ then
    let $cond = cond \wedge$ valid_address $(r_5 + r_6)\ m$ in
    let $r_4 = m(r_5 + r_6)$ in
    let $cond = cond \wedge$ word_aligned_address $r_4$ in
    let $(pc, r_{14}) = (r_4, \texttt{L2})$ in
      $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
  else if $pc = \texttt{L2}$ then
    let $r_6 = r_6 - 4$ in
      if $r_6 = 0$ then $(cond, \texttt{L4}, r_4, r_5, r_6, r_{14}, m)$
                  else $(cond, \texttt{L0}, r_4, r_5, r_6, r_{14}, m)$
  else $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
```

The automatically derived certificate theorem makes use of a feature of our machine-code Hoare triple that allows the pre- and postconditions to mention the value of the program counter (PC) as state component, i.e. control does not need to enter/exit at specific points of the code in the Hoare triple.

$(\text{calls}(c, pc, r_4, r_5, r_6, r_{14}, m) = (\text{true}, pc', r_4', \ldots)) \Longrightarrow$
{ ARM state holds $(r_4, r_5, r_6, r_{14}, m)$ and PC is pc }
`E7954006 E12FFF34 E2566004 1AFFFFFB`
{ ARM state holds $(r_4', r_5', r_6', r_{14}', m')$ and PC is pc' }

With this result from decompilation one can verify properties of this code without tedious proofs in the Hoare logic.

III. Improved Decompilation Algorithm

The new decompilation algorithm has three phases. The key technical differences over our previous approach [13] are highlighted with **bold text**.

Phase 1: Evaluate the underlying ISA model for each machine instruction; derive a theorem, **stated in terms of a simple machine-code Hoare triple**, describing each instruction; and in order to make *phase 3* faster, also **make the code segment of each Hoare triple identical** (explained in Section III-A).

Phase 2: Compute the control-flow graph (CFG) of the given code using information gathered from the Hoare triples.

ARM machine code	instr.	time/cost of old	time/cost of new	reduction	model eval.
sum of array (Sec. I-A)	4	2.5 s (73,039 i)	0.3 s (4,019 i)	86 % (94 %)	7.8 s (1.5 Mi)
copying garbage collector [10]	89	50 s (1,526,281 i)	6.0 s (53,301 i)	88 % (97 %)	173 s (40 Mi)
1024-bit multiword addition	224	70 s (1,029,685 i)	1.2 s (10,802 i)	98 % (99 %)	37 s (8.9 Mi)
256-bit Skein hash function	1,352	5,366 s (21,432,926 i)	56 s (1,842,642 i)	99 % (91 %)	500 s (105 Mi)

Fig. 1. Benchmarks comparing the new approach (new) with our previous approach (old). The cost is given in seconds (s) and number of primitive higher-order logic inferences (i) in HOL4 [16]. The cost of evaluating the ISA model is separated as this cost is independent of decompilation approach.

Split the CFG into separate *decompilation rounds*, i.e. separate inner loops from enclosing outer loops where possible. For complicated CFGs, **insert an extra final decompilation round which ties up the disjoint pieces if necessary** (as illustrated by the example in Section II-A).

Phase 3: For each decompilation round: compose the Hoare triples following the CFG **in a way which directly constructs the extracted function in the postcondition of the theorem** (Section III-B). **This function in the postcondition also collects accumulated side conditions as if they were updates to a state component** (*cond* in Section III-A). If the code has a loop, a loop rule is applied which wraps the result up using a tailrec-combinator and **combines the resulting side condition on termination with the other side conditions**.

A. Simple Machine-Code Hoare Logic

The machine-code Hoare triples, $\{\,pre\,\}\ code\ \{\,post\,\}$, that were used above will be explained next. More formally, these are parametrised by two functions: $next$, a the next-state function for the ISA model of interest; and $assert$, a state assertion which inspects the state (explained below).

$$(assert, next)\ \vdash\ \{\,pre\,\}\, code\, \{\,post\,\} \qquad (1)$$

This machine-code Hoare triple is defined to be true: if for all states that satisfy pre and including $code$, then another state can be reached (by some n applications of $next$) such that $post$ is true for this state and $code$ is included in it. The total-correctness Hoare triple (1) is formally defined to mean,

$$\forall state\ c.\ assert\ (code \cup c, pre)\ state \implies$$
$$\exists n.\ assert\ (code \cup c, post)\ (next^n(state))$$

where the set union $\cup$ with arbitrary code extension c is present to facilitate extension of the code (explain in the next section).

We instantiate $next$ and $assert$ for each supported architecture, e.g. ARM, x86 or PowerPC. We instantiate $assert$ to check that each state component is consistent with $code$ and pre/$post$. Here $code$ is represented as a set of (address,instruction) pairs, and pre and $post$ are, for efficiency reasons, simply a large tuple listing the value of state components, e.g. $(pc, r_0, r_1, \ldots)$ asserts that the value of PC is pc and register 0 is r_0 etc. By representing pre/$post$ as tuples, composition and matching becomes fast and simple. We always include a special $cond$ element in $assert$. This $cond$ is a condition for the entire assertion to make sense, e.g. for ARM we instantiate $assert$ with:

$$\mathsf{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state =$$
$$(cond \implies code \text{ is in memory of } state \text{ and}$$
$$\text{the PC of } state \text{ is } pc \text{ and } \ldots)$$

{ARM registers `r1-r3` are (r_1, r_2, r_3) and
m is a model of part of memory and PC is `L0`}
`E7921003 E0800001 E2533004 1AFFFFFB`
{ARM registers `r1-r3` are $(m(r_2 + r_3), r_2, r_3)$ and
m is a model of part of memory and PC is `L1` and
$\mathsf{valid\_address}(r_2 + r_3)\ m$ is added to $cond$}

{ ARM assert $(c, r_0, r_1, r_2, r_3, m)$ and PC is `L1` }
`E7921003 E0800001 E2533004 1AFFFFFB`
{ let $(pc', c', r_0', r_1', r_2', r_3', m') =$
(let $r_0 = r_0 + r_1$ in
let $r_3 = r_3 - 4$ in
if $r_3 = 0$ then $(\mathtt{L4}, c, r_0, r_1, r_2, r_3, m)$
else $(\mathtt{L0}, c, r_0, r_1, r_2, r_3, m)$) in
ARM assert $(c', r_0', r_1', r_2', r_3', m')$ and PC is pc' }

Fig. 2. Two machine-code Hoare triples for: (a) the load instruction from Section I-A, and (b) the last three ARM instructions from Section I-A. Both contain other code too, explained in Section III-B.

B. Composing Hoare triples

Our machine-code Hoare triple supports composition:

$$\{pre\}\, code\, \{m\} \wedge \{m\}\, code\, \{post\} \implies \{pre\}\, code\, \{post\}$$

For this rule to be applicable, the Hoare triples must have identical code sets $code$. Note that each code set is a set of (address,instruction) pairs which is a *sufficient* assumption for getting execution from pre to $post$. To make the code sets identical, we apply the following theorem which can be used to extend the code sets. This theorem is applied as a pre-processing step in *Phase 1* to speed up composition in *Phase 3*. Here $\subseteq$ is the ordinary subset relation.

$$\{pre\}\, code_1\, \{post\} \wedge code_1 \subseteq code_2 \implies \{pre\}\, code_2\, \{post\}$$

In *Phase 3*, composition of Hoare triples is performed bottom-up following the CFG (or the part of it which is relevant for this decompilation round). Each compositions returns a theorem where the relevant part of the extracted function, including the side conditions, appears in the postcondition. Each composition returns a theorem of the form:

$$\begin{array}{l}\{pre[v_0 \ldots v_n]\}\\ code\\ \{\text{let } (v_0' \ldots v_n') = f(v_0 \ldots v_n) \text{ in } post[v_0' \ldots v_n']\}\end{array} \qquad (2)$$

Figure 2 show the concrete inputs to the final composition for the sum-of-an-array example (Section I-A). The second input carries the extracted function in the form of (2).

C. Extracting Recursive Functions

As illustrated by our first example in Section I-A, loops in the machine code turn into loops in the extracted function (if the control-flow is simple enough). We define these tail-recursive functions by instantiating f, g and d in the following function template:

$$\mathsf{tailrec}\ g\ f\ d\ x = \mathsf{if}\ g\ x\ \mathsf{then}\ \mathsf{tailrec}\ g\ f\ d\ (f\ x)\ \mathsf{else}\ d\ x$$

Our definition of tailrec is based on while $g\ f\ x$ which repeatedly applies f to x until g becomes false. Crucially, while can be defined in (higher-order) logic without a termination proof [7], which is important because most of the functions the decompiler extracts do not terminate for all inputs. However, note that our Hoare triples are total-correctness Hoare triples, i.e. we need to know that our use of while terminates for certain inputs: it terminated for input x if $\neg g\ (f^n\ x)$. For this reason, we insert the termination requirement into the definition of tailrec. We make this requirement part of the $cond$ side condition that our extracted functions produce:

$$\begin{array}{l}\mathsf{tailrec}\ g\ f\ d\ x = \\ \quad \mathsf{let}\ (cond, v_1 \ldots v_n) = d\ (\mathsf{while}\ g\ f\ x)\ \mathsf{in} \\ \qquad (cond \wedge (\exists n.\ \neg g\ (f^n\ x)), v_1 \ldots v_n)\end{array}$$

Any verification that uses such extracted function must prove that the returned $cond$ is equal to true (for relevant input values); otherwise, the postcondition of the certificate theorem has no meaning (due to $\Longrightarrow$ at the bottom of Section III-A).

To introduce a tail-recursive function, we apply the following theorem with appropriate instantiations of pre, $post$ etc.

$$\begin{array}{l}(\forall x.\ \{pre\ x\}\ code\ \{\mathsf{if}\ g\ x\ \mathsf{then}\ pre\ (f\ x)\ \mathsf{else}\ post\ (d\ x)\} \\ \Longrightarrow \\ (\forall x.\ \{pre\ x\}\ code\ \{post\ (\mathsf{tailrec}\ g\ f\ d\ x)\})\end{array}$$

Often pre and $post$ are instantiated with functions that simply just set the program counter value. For the example mentioned above, pre is instantiated as follows to set the PC to L0.

$$\begin{array}{l}\lambda(c, r_0, r_1, r_2, r_3, m). \\ \quad \text{ARM state holds } (r_0, r_1, r_2, r_3, m, c) \text{ and PC is } \mathtt{L0}\end{array}$$

In our implementation, we avoid defining separate component functions f, g and d by defining a single function which returns three different outcomes. We omit the details of this space optimisation as it is not crucial for understanding the main novelties of the new technique: the new Hoare triple; collection of side conditions as if they were state updates; and our new approach to handling complicated control flow.

IV. Summary and Related Work

This paper has presented significant improvements to decompilation into logic — a technique which aids verification of machine-code programs. We have simplified the technique, optimised it for mechanisation speed and made it applicable even to code with arbitrary use of code pointers.

Formal verification of machine code using theorem provers was pioneered in impressive work by Boyer and Yu [2]. Boyer and Yu verified functional correctness of string functions compiled for the Motorola MC68020. Their proofs were carried out in the Boyer-Moore theorem prover Nqthm [6] and required significant manual effort. Since then most work in this area has focused on making proofs easier: Matthews et al. [8] have showed how verification condition generation for machine code can be accomplished, Hardin et al. [5] show how ACL2 can be used and our work [13] has shown how functions can be extracted from machine code and how that aids verification. Various program logics for assembly and machine code have also been developed [10], [15], [1], [3]. Chlipala's approach [3], using Coq, has a distinct emphasis on proof automation for functional correctness. There has also been work targeting mostly automatic proofs of basic safety properties for low-level code [18], [9], [17].

References

[1] Nick Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL)*, Computer Science Logic. Springer, 2006.

[2] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[3] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[4] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2010.

[5] David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, ACL2 '06, pages 11–20, New York, NY, USA, 2006. ACM.

[6] M. Kaufmann, R. S. Boyer, and J Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[7] Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.

[8] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Logic Programming and Automated Reasoning (LPAR)*. Springer, 2006.

[9] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Principles of Programming Languages (POPL)*. ACM, 1998.

[10] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

[11] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2011.

[12] Magnus O. Myreen, Thomas Sewell, Michael Norrish, and Gerwin Klein. Using the Cambridge ARM model to verify the concrete machine code of seL4. Talk at HCSS'11 http://cps-vo.org/node/1127, 2011.

[13] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2008.

[14] Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In de Moor and Schwartzbach, editors, *Compiler Construction (CC)*, LNCS. Springer, 2009.

[15] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. *ACM SIGPLAN Notices*, 41(1):320–333, January 2006.

[16] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[17] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer, 2006.

[18] Lu Zhao. *A program logic and its applications to fully verified software fault isolation*. PhD thesis, University of Utah, 2012.

Complete and Effective Robustness Checking by Means of Interpolation

Stefan Frehse[1] Görschwin Fey[1,3] Eli Arbel[2] Karen Yorav[2] Rolf Drechsler[1,4]

[1]Institute of Computer Science
University of Bremen, Germany
sfrehse@informatik.uni-bremen.de

[2]IBM Research Labs
Haifa, Israel
{arbel,yorav}@il.ibm.com

[3]Institute of Space Systems
German Aerospace Center
Bremen, Germany
goerschwin.fey@dlr.de

[4]Cyber-Phsyical Systems
DFKI-GmbH
Bremen, Germany
rolf.drechsler@dfki.de

Abstract

Technology scaling continues to downscale feature sizes. As a side-effect this has some serious drawbacks, in particular increasing vulnerability of circuits against transient faults caused, e.g., by radiation. Even under malfunctions of internal components the circuit must behave as specified. Several techniques have been proposed to overcome this problem. However, the implementation of those techniques in the design might be buggy and needs to be verified.

This paper provides an effective algorithm using formal reasoning to completely analyze the fault tolerance of a circuit, under all input sequences and all transient faults. The algorithm based on interpolation identifies components in which transient faults are observable. Experiments show that the newly introduced complete approach analyzes ITC'99 and IBM circuits, effectively.

I. Introduction

Technology scaling decreases power consumption and increases the integration density as well as computational throughput. As a side-effect technology scaling comes inherently with serious problems. In particular vulnerability against transient faults increases, caused by radiation or process variation. Transient faults are modeled as the negation of a signal at logic level, i.e, switching from 0 to 1 or 1 to 0 for a short period of time. However, the circuit must behave as specified even under those internal faults. Various techniques to catch and handle those faults are available. These techniques operate on different levels of the design flow. At design level, techniques such as *Triple-Modular-Redundancy* (TMR) or Hamming-Code are applied for detection and correction of internal faults. Furthermore, application specific techniques are available [1]. However, the implementation of these techniques in the design may be faulty itself and needs to be verified to ensure correctness or to show vulnerable parts of the circuit [2].

This work was supported in part by the European Union (Project DIAMOND, FP7-2009-IST-4-248613).

To ensure correctness of the implementation *complete robustness checking* must be performed: 1) under all possible input sequences and 2) under all transient faults it has to be proven that 3) all outputs sequences adhere to the specification.

Formal approaches have been proposed, which analyze the impact of transient faults on the functional output of the circuit. Previous complete approaches [3], [4], [5], [6] analyzing sequential circuits relying on *Binary Decision Diagrams* are restricted to small circuits due to the state explosion problem. The approach of [7] computes the vulnerability against a user-defined specification in terms of a property based on model checking techniques. Approaches based on *Boolean Satisfiability* (SAT) are either restricted to self-checking circuits (e.g. [8]) or consider only short time intervals [9].

Bounded Model Checking (BMC) based on SAT has been significantly improved by interpolation [10], which is further enhanced in, e.g., [11], [12], [13], [14], [15]. The approach proves or disproves a safety property for finite systems by a fixed-point computation and is successfully applied in complex hardware verification in industry.

Here, we propose two approaches for robustness checking: First, SAT-based robustness checking is improved by utilizing interpolation similarly to interpolation in BMC. Second, we go one step beyond by over-approximating the entire set of reachable states also based on interpolation. This over-approximation can be utilized in BMC for a series of safety properties in general. In robustness checking this is required to identify components that may cause *Silent Data Corruption* (SDC) upon failure - called *unbounded dangerous components* in the following. Knowing such components is mandatory as any second fault may corrupt the system behavior. Previous SAT-based approaches for robustness checking cannot identify unbounded dangerous components. Experimental results on hard benchmarks show the effectiveness in comparison to a previous approach.

The paper is structured as follows: Section II covers the preliminaries in particular the fault model and the computational model. In Section III, the approach adapting interpolation-based model checking for robustness checking is presented. Section IV describes the complete approach including the over-approximation of reachable states and the fixed-point iteration based on interpolation. Section V provides the evaluation of both approaches and Section VI concludes.

II. Preliminaries

A. Circuits and Transition Systems

A synchronous circuit $C = (V, E, L)$ is a directed graph with vertices (components) V, edges $E \subseteq V \times V$, and a labeling function $L : V \to \{\mathrm{IN}, \mathrm{FF}, \mathrm{AND}, \mathrm{NOT}\}$, which maps each vertex to a function. The vertices labeled with FF are state elements. A *state* of the circuit is represented by the values of the FF nodes: Each FF $v_{\mathrm{FF}} \in V$ is mapped to a Boolean value, i.e., $v_{\mathrm{FF}} \mapsto \mathbb{B}$. From a circuit C a transition system $M = (I, S, T)$ is derived. The set $I \subseteq S$ describes the *initial* states. The state space of a circuit with m FFs is given by $S = \mathbb{B}^m$. The transition relation $T(s, s')$ is true, if there is a transition from present state s to a next state s'. The set $\mathsf{img}(Q) = \{s' \in S \mid \exists s \in Q \wedge T(s, s')\}$ contains all successor states reachable in one step from the states in $Q \subseteq S$. The operator img is called an exact image operator. Let $\mathsf{img}(Q)^0 = Q$ and $\mathsf{img}^{i+1}(Q) = \mathsf{img}(\mathsf{img}^i(Q))$, all states reachable from I are given by: $S^* = \bigcup_{i \geq 0} \mathsf{img}^i(I)$. The over-approximation operator $\hat{\mathsf{img}}$ has the following properties: $\mathsf{img}(Q) \subseteq \hat{\mathsf{img}}(Q)$ for $Q \subseteq S$ and hence it holds $S^* \subseteq \hat{S}$ with $\hat{S} = \bigcup_{i \geq 0} \hat{\mathsf{img}}^i(I)$.

In this paper a set of states S is often described by a Boolean predicate δ. We say *the set* δ is an abbreviation for $S = \{s | \delta(s) = 1\}$. Both symbols are used interchangeably.

A formula is called *concrete* if the formula does not contain any approximation operators. Otherwise, the formula is called *abstract*.

B. Boolean Satisfiability & Interpolation

Given a Boolean function, the Boolean satisfiability problem (SAT-problem) is to decide whether there exists an assignment such that the function evaluates to true. If there exists such an assignment, the formula is called *satisfiable* otherwise *unsatisfiable*. Often a *Conjunctive Normal Form* (CNF) is given as input to a SAT solver. A CNF is a conjunction of clauses, where a clause is a disjunction of literals. A literal is a variable x or its negation $\neg x$. A CNF formula F is a set of clauses $F = \{c_1, \ldots, c_k\}$, whereas a clause is a set of literals $c_i = \{l_1, \ldots, l_m\}$. The set of variables of a formula F is denoted as $\mathsf{Var}(F)$.

Given a pair of propositional formulas (A, B) such that $A \wedge B$ is unsatisfiable, there exists an interpolant σ of (A, B) with the following properties based on *Craig's Interpolation Theorem* [16]: 1) σ is a propositional formula over the subset of common variables of A and B, i.e., $\mathsf{Var}(\sigma) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$, 2) A implies σ, and 3) $B \wedge \sigma$ is unsatisfiable. Intuitively, this means, σ abstracts some facts of A while σ contradicts B. Given a resolution proof of an unsatisfiable CNF an interpolant is computed by, e.g., [10], [17], [18], [19] in linear time with respect to the size of the proof. Note, the size of the proof may be exponentially larger than $|A \cup B|$. Let $\mathsf{itp}(A, B)$ be a function that computes an interpolant of the unsatisfiable pair (A, B).

C. Robustness Checking

The goal of robustness checking is to identify parts of a circuit that may cause unwanted behavior under occurrences of transient faults or to prove that any fault of a component is detected. We consider gates in this work to simplify the presentation. By considering more complex modules as components, multiple faults can be modeled as well [9]. Our approach can easily be lifted to component level, too.

A transient fault is modeled as a non-deterministic bit flip of an output of a node for one time frame. Furthermore, suppose the circuit is equipped with a fault signal f, which reports detected faults of the system by setting f to one. If the system does not have a fault signal, f is equal to zero is assumed. Components of the circuit are classified based on the behavior of the system when a fault is injected, i.e., a component $v \in C$ belongs to one of three classes: **non-robust**: at least one fault at node v is observable after any number of time frames at the primary outputs and no fault is reported, i.e., $f = 0$, **dangerous**: any fault at v only modifies the state after any number of time frames but is not observable on the primary outputs and $f = 0$, or **robust**: otherwise, i.e., all faults are either masked and not observable at the primary outputs or they are reported by the fault signal. Let $\mathbb{T}$ be the set of robust components, $\mathbb{S}$ the set of non-robust components and $\mathbb{D}$ the set of dangerous components: $V = \mathbb{T} \cup \mathbb{S} \cup \mathbb{D}$.

Given a circuit under verification $C = (V, E, L)$, the circuit $C_D(V', E', L')$ with $D \subseteq V$ is constructed by inserting a multiplexer at the component's output for each component $v \in D$ and the primary inputs of C and C' are stimulated by the same values. Each new multiplexer has a selector variable denoted by a_v for a component $v \in D$. The *data 1*-input of the multiplexer is a fresh primary input and if the selector variable a_v is activated, an arbitrary value is chosen for this input. Otherwise, the normal computation of the component is performed, based on the *data 0*-input connected to the output of the respective component. This construction allows for a non-deterministic bit flip if one selector variable a_v is activated. The derived transition system of C_D is given by $M_D = (I, S', T')$ where the transition relation T' is extended as follows: $T_D = T' \wedge \left(\sum_{v \in D} a_v = 1\right)$, where $\sum$ adds Boolean variables, i.e., exactly one fault injection is performed. Formulas for determining non-robust components are constructed as follows:

$$\mathsf{init}(l) = I(x_0) \wedge \bigwedge_{i=1}^{l} T(x_{i-1}, x_i) \tag{1}$$

$$\mathsf{inj}(l, D) = T(x_l, x_{l+1}) \wedge T_D(x_l, \dot{x}_{l+1}) \tag{2}$$

$$\begin{aligned}\mathsf{prop}_{\mathsf{NR}}(l, k) = &\bigwedge_{i=l+1}^{l+k} (T(x_i, x_{i+1}) \wedge T(\dot{x}_i, \dot{x}_{i+1})) \\ &\wedge P(x_{l+k+1}, \dot{x}_{l+k+1})\end{aligned} \tag{3}$$

where $P(x_{l+k+1}, \dot{x}_{l+k+1})$ forces the primary outputs to be different in the last time frame. The fault signal is forced to be zero to consider scenarios where malfunctions are not detected. However, this is not explicitly written in the formula, to keep the formulas simple. Formula (1) is satisfiable if and only if there is at least one path of length l from the initial state x_0 to a state x_l. Formula (2) computes a normal transition from x_l to x_{l+1} based on the transition relation T and a transition from x_l to $\dot{x}_{l+1}$ based on T_D, i.e., a single fault is **injected** at a component $v \in D$. Finally, in Formula (3), the correct state x_{l+1} and the faulty state $\dot{x}_{l+1}$ are **propagated** over k time frames and the primary outputs are checked for equivalence at the last time frame expressed by P. Conjoining these three formulas the entire formula for determining non-robust components is obtained:

$$\phi(l, k, D) = \mathsf{init}(l) \wedge \mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathsf{NR}}(l, k) \tag{4}$$

Lemma 1. *Given a non-empty set of components to classify $D \subseteq V$. If the formula $\phi(l, k, D)$ is satisfiable for an l and k, with $a_v = 1$ for a component $v \in D$, then the component v is non-robust.*

Similar to Formula 4 dangerous components are computed using the following formula that compares the states in the last time frame. Here, only the last part is different from Forumla 4 which is given by:

$$\begin{aligned}\mathsf{prop}_{\mathsf{D}}(l, k) = &\bigwedge_{i=l+1}^{l+k} (T(x_i, x_{i+1}) \wedge T(\dot{x}_i, \dot{x}_{i+1})) \\ &\wedge (x_{l+k+1} \neq \dot{x}_{l+k+1})\end{aligned} \tag{5}$$

and the entire formula for classifying dangerous components is given by:

$$\psi(l, k, D) = \mathsf{init}(l) \wedge \mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathsf{D}}(l, k)$$

The function $\mathsf{sol}(\phi(l, k, D))$ computes all non-robust components of D and the function $\mathsf{sol}(\psi(l, k, D))$ computes all dangerous components of D with respect to the values of l and k. For each combination of l and k non-robust components are determined and afterwards the dangerous components are determined.

In order to compute the complete set of non-robust components of the circuit under verification each combination of $l \in \{0, \ldots, l'\}$ and $k \in \{0, \ldots, k'\}$ has to be checked, where l' and k' are sufficiently large completeness thresholds, i.e.,

$$\mathbb{S} = \bigcup_{\substack{l \in \{0,\ldots,l'\} \\ k \in \{0,\ldots,k'\}}} \mathsf{sol}(\phi(l, k, V)).$$

Once all combinations of l and k have been checked, all non-robust components are determined. While checking all combinations of l and k dangerous components are computed by comparing state bits instead of primary outputs. SDC may occur, i.e., components remain classified as dangerous even when the thresholds for l and k are reached. That means, under all input sequences, any possible faults is not observable at any time on the primary outputs, but at least one fault corrupts the state. This behavior is classified as **unbounded dangerous** and components showing this behavior are in the set $\mathbb{U} \subseteq V$. Given all non-robust components and unbounded dangerous components, the robust components are given by $\mathbb{T} = V \setminus (\mathbb{S} \cup \mathbb{U})$. However, the completeness thresholds might be very large, which makes it hard or practically impossible to check all combinations. The threshold for l is given by the diameter of the transition system $M(I, S, T)$, i.e., $l' = \mathrm{dia}(M)$, because all reachable states can be reached within this number time frames. A completeness threshold for k is given by $k' = 2^{2m}$ with m FFs nodes, because all states of the product machine of the normal transition system and the transition system with fault injection have to be discovered.

III. BMC with Interpolation for Robustness Checking

Checking every combination of l and k is infeasible in practice. However, a complete answer can often be given before reaching l' by applying interpolation-based model checking [10] for robustness checking. Interpolation is exploited in order to a find termination criterion before unrolling the transition relation l' times by computing a fixed-point.

A safety-property holds on a circuit if it is proven that the property holds in every reachable state of the circuit's automaton. BMC has been introduced to check the property on states by iteratively unrolling the transition relation. All reachable states are checked when the transition relation is unfolded l' times. This may become very expensive for larger circuits. Interpolation-based model checking [10] often terminates before reaching l'. Interpolants abstract some facts which are irrelevant to prove the property and therefore speeds up the convergence to a fixed-point. In order to prove a property only relevant states are considered.

In robustness checking a series of safety properties for all $k \in \{0, \cdots, k'\}$ has to be checked. For each new property a model checking instance is solved determining the set of non-robust components under relevant reachable states. Mapping robustness checking to interpolation-based model checking, avoids considering all values of $l \in \{0, \ldots, l'\}$. That means, for each k an earlier termination may be reached.

In the following, interpolation-based model checking [10] is adapted for robustness checking. Given $\phi(l, k, D)$ as defined in Equation 4, the property to check is composed of injection, propagation, and forcing the primary outputs to be different: $\mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathrm{NR}}(l, k)$. Starting with $l = 0$, consider a non-empty set of components to classify $D \subseteq V$ and the formula pair (A, B) with:

$$A := I(x_0) \wedge T(x_0, x_1) \tag{6}$$

$$B := \bigwedge_{i=2}^{l} T(x_{i-1}, x_i) \wedge \overbrace{\mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathrm{NR}}(l, k)}^{\text{property of length } k+1} \tag{7}$$

Suppose all non-robust components have been determined with respect to the values of l and k and only the remaining components not shown to be non-robust (at least one, i.e.,$|D| \geq 1$) are extended by a multiplexer to inject a fault. In this case, $A \wedge B$ is unsatisfiable. An interpolant σ with $\sigma = \mathsf{itp}(A, B)$ is computed based on the resolution proof of the SAT solver. The interpolant σ is defined over the state variables expressed by x_1, i.e, the common variables of A and B with $\mathsf{Var}(\sigma) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$.

The part B may be unsatisfiable itself in circuits containing checker circuitry: The fault signals is constrained to zero but any injection of a fault forces the fault signal to one. In this case, determining the interpolant is skipped because all reachable states are covered.

Otherwise, the interpolant is added to A such that $A = (I(x_0) \vee \sigma(x_0)) \wedge T(x_0, x_1)$ where the variable x_1 of σ is replaced by x_0 and the procedure restarts. A fixed-point is reached when the disjunction of all previously computed interpolants implies the new interpolant. Once the instance becomes satisfiable on such an abstract formula, a potentially spuriously classified non-robust component has been determined due to the over-approximation by the interpolant. Then, the procedure restarts by increasing the value of l, which either allows for classifying non-robust components on the concrete formula or makes the interpolants weaker by strengthening B. A fixed-point will be eventually found and the classification is complete with respect to the current value of k [10]. After reaching a fixed-point, k is increased by one and the interpolation-based model checking procedure restarts, discarding all interpolants.

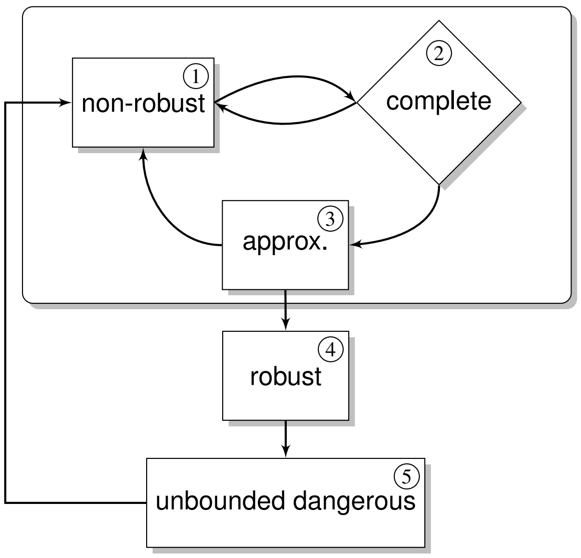
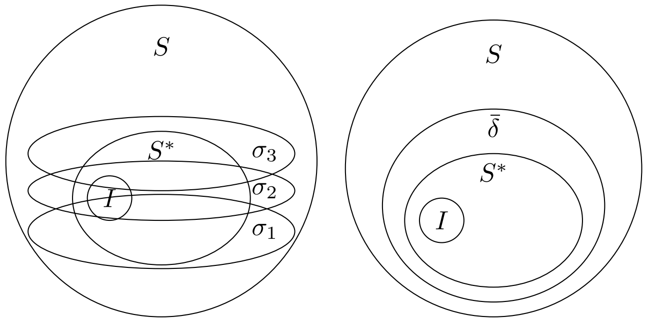


Fig. 1: Flow of new approach

IV. Complete Classification

The approach from the previous Section III may find earlier termination criterion for each k. However, calling this approach for all values of k still remains infeasible. A new approach presented in this section may terminate earlier than reaching l' and k'.

A rough overview of the new approach is depicted in Figure 1: Step ① classifies non-robust components. The subsequent Step ② checks whether all relevant reachable states are considered for a fixed k. These steps have been described in the previous Section III and are embedded in this flow. Any classification of non-robust components is based on reachable states. If there are more states to be covered, the transition relation is further unrolled and the flow goes back to Step ①. Otherwise, if all relevant states are considered an over-approximation of the entire set of reachable states based on a newly introduced construction of interpolants is performed in Step ③ and is described in Section IV-A. After having a new over-approximation, non-robust and dangerous components are over-approximated. This classification is exploited in order to compute robust components. All dangerous components are considered for a further analysis in Step ④, because these components are potentially unbounded dangerous. For example, this



(a) Interpolation by McMillan: (σ) (b) Adequate Approximation: (δ)

Fig. 2: Approximations by Interpolation

occurs in TMR circuits: a fault is corrected by the majority voter but the affected module remains in an erroneous state. In order to prove that these components are unbounded dangerous the proof procedure is called in Step ⑤ and is presented in Section IV-C.

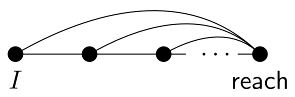
A. Over-approximation of Reachable States

The termination before reaching k' requires that at least all reachable states are modeled for fault injection. Since computing the exact set of reachable states is hard, an over-approximation is introduced in the following based on interpolation. The difference of the interpolants from Section III ([10]) and those here is illustrated in Figure 2. In both figures S is the complete state space and S^* is the set of states reachable from the initial state as introduced in Section II. The left figure shows interpolants σ_1, σ_2, and σ_3 after reaching a fixed-point in three steps. The union of all interpolants yields an over-approximation of the exact set of reachable states. In order to compute a fixed-point using interpolation-based model checking at least one step of interpolation is required, often a few steps, in order to reach a fixed-point. The right figure shows an interpolant $\bar{\delta}$ from the new approach. The computation of this interpolant requires exactly one step of interpolation with additional model checking steps.

Both 1) the union of all interpolants from McMillan's approach and 2) the interpolant introduced in this work over-approximate the exact set of reachable states but the quality, i.e., the number of included non-reachable states may differ. Due to space limitation a detailed comparison about differences cannot not be discussed.

The computation of the over-approximation is introduced in the following. A new partition of (A, B) and a check whether the computed interpolant is an adequate approximation are introduced.

To compute the over-approximations, interpolants are derived from the formula $\mathsf{reach}(l)$ which models all states



reachable in l steps from the initial state illustrated in the figure on the right hand side.

$$\mathsf{reach}(l) = I(x_0) \wedge \bigwedge_{i=1}^{l} (I(x_i) \vee T(x_{i-1}, x_i)) \qquad (8)$$

All reachable states are modeled on x_l, if $l \geq l'$. The entire formula to determine non-robust components becomes:

$$\phi_{\mathsf{reach}}(l, k, D) = \mathsf{reach}(l) \wedge \mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathrm{NR}}(l, k) \quad (9)$$

The difference between $\phi(l, k, D)$ and $\phi_{\mathsf{reach}}(l, k, D)$ is that the state x_l for injection might be any state along any path of length l from the initial state rather than only states reachable in l steps. Based on this formulation an over-approximation of the exact set of reachable states is derived.

Definition 1. *Given a transition system $M = (I, S, T)$ and a predicate δ defined over the state variables of M. Then δ is an* **adequate approximation** *if the set δ contains only non-reachable states.*

Lemma 2. *Given an adequate approximation δ, then for all $s \in S^*$, $\bar{\delta}(s)$ is true. That means $\bar{\delta}$ is an over-approximation of the reachable states.*

In order to compute adequate approximations consider the following pair (A', B') of formulas for given l and k

$$\phi_{\mathsf{reach}}(l, k, D) = \overbrace{\mathsf{reach}(l)}^{B'} \wedge \overbrace{\mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathrm{NR}}(l, k)}^{A'} . \qquad (10)$$

The interpolant $\delta = \mathsf{itp}(A', B')$ computes states that are not reachable from the initial state in l or less steps, but A' implies δ, i.e., states in δ fulfill the property. These states are possibly reachable from the initial state in more than l steps or are non-reachable states.

Theorem 1. *Given a transition system $M = (I, S, T)$. There exists an $l \leq l'$ and an $k \leq k'$ with a non-empty set $D \subseteq V$ of components proven to be not non-robust with respect to l and k. Then, $\phi_{\mathsf{reach}}(l, k, D)$ is unsatisfiable and $\delta = \mathsf{itp}(A', B')$ is an adequate approximation.*

Proof: Setting $l = l'$ and $k = k'$, $\mathsf{reach}(l)$ models all reachable states and $\phi_{\mathsf{reach}}(l, k, D)$ is unsatisfiable with $|D| \geq 1$. An interpolant $\delta = \mathsf{itp}(A', B')$ is computed. All states for which A' is true satisfy δ. These states are only non-reachable states since $\delta \wedge B'$ is unsatisfiable and B' models all reachable states for $l = l'$. ■

This proves that an adequate over-approximation is computed when setting l to l'. However, in practice an adequate approximation is often computed before l reaches l'. In order to verify that the computed interpolant δ is an adequate approximation, a separate model checking step is performed, i.e., the interpolant is checked for reachable states. We employ an interpolation-based model checker to check the invariant $\bar{\delta}$.

While checking each combination of l and k to determine non-robust components using $\phi_{\mathsf{reach}}(l,k,D)$, interpolants are computed based on Theorem 1. If an interpolant is an adequate approximation, the interpolant is added to the set Δ, where Δ contains all adequate approximations.

B. Approximation of Robust Components

Given a set of adequate approximations $\Delta = \{\delta_1, \dots, \delta_n\}$ non-robust and dangerous components are over-approximated using the following formulas. All remaining components are robust components.

At first, the formula for over-approximating non-robust components is constructed:

$$\hat{\phi}(l,k,D,\Delta) = \bigwedge_{\delta\in\Delta} \bar{\delta}(x_l) \wedge \mathsf{inj}(l,D) \wedge \mathsf{prop}_{\mathsf{NR}}(l,k). \tag{11}$$

Lemma 3. *Given a non-empty set $D \subseteq V$ and a set of adequate approximations Δ. Let $\mathbb{S}_l^k = \mathsf{sol}(\phi(l,k,D))$ and $\hat{\mathbb{S}}_l^k = \mathsf{sol}(\hat{\phi}(l,k,D,\Delta))$, then $\mathbb{S}_l^k \subseteq \hat{\mathbb{S}}_l^k$ is true for any l and k. That means, $\hat{\mathbb{S}}_l^k$ is an over-approximation of non-robust components.*

Since the formula $\hat{\phi}$ may consider more states that ϕ because an over-approximation is constrained and thus non-robust components are over-approximated.

Next, an over-approximation of the set of dangerous components is introduced. As described in Section II, a fault injected into a dangerous component corrupts the state but is not observable at the primary outputs. The formula to over-approximate dangerous components is given by:

$$\hat{\psi}(l,k,D,\Delta) = \bigwedge_{\delta\in\Delta} \bar{\delta}(x_l) \wedge \mathsf{inj}(l,D) \wedge \mathsf{prop}_{\mathsf{D}}(l,k) \tag{12}$$

Lemma 4. *Given a non-empty set $D \subseteq V$ and a set of adequate approximations Δ. Let $\mathbb{D}_l^k = \mathsf{sol}(\psi(l,k,D))$ and $\hat{\mathbb{D}}_l^k = \mathsf{sol}(\hat{\psi}(l,k,D,\Delta))$, then $\mathbb{D}_l^k \subseteq \hat{\mathbb{D}}_l^k$ is true for any l and k. That means, $\hat{\mathbb{D}}_l^k$ is an over-approximation of dangerous components.*

The over-approximated sets of non-robust and of dangerous components determine a subset of robust components as stated in the following lemma.

Lemma 5. *Given an over-approximated set of non-robust $\hat{\mathbb{S}}_l^k$ and dangerous $\hat{\mathbb{D}}_l^k$ components, respectively. A set of robust components is given by: $\check{\mathbb{T}}_l^k = V \setminus (\hat{\mathbb{S}}_l^k \cup \hat{\mathbb{D}}_l^k)$.*

By checking all combinations of l and k the final result of non-robust and robust components is determined, i.e., $\mathbb{S} = \bigcup_{l,k} \mathbb{S}_l^k$ and $\mathbb{T} = \bigcup_{l,k} \check{\mathbb{T}}_l^k$. Reaching l' and k', the classifications are complete.

However, components are unbounded dangerous on certain circuits even when the thresholds for l and k are met. All these unbounded dangerous components are reconsidered in a next iteration. In order to prove that these components are unbounded dangerous, it is required to prove that any fault of these components will not affect the primary outputs for any combination of l and k. The following section provides the corresponding proof procedure.

C. Fixed-point Computation on the Property

Suppose arbitrary values for l and $k > 0$, and a non-empty set of components $D \subseteq V$ which are to be proven or to be refuted to be unbounded dangerous components. Formula (11), $\hat{\phi}(l,k,D)$ is unsatisfiable, therefore an interpolant of $\sigma = \mathsf{itp}(X,Y)$ can be computed where:

$$X := \bigwedge_{\delta\in\Delta} \bar{\delta}(x_l) \wedge \mathsf{inj}(l,D) \wedge T(x_{l+1},x_{l+2}) \wedge T(\dot{x}_{l+1},\dot{x}_{l+2}) \tag{13}$$

$$Y := \bigwedge_{i=l+2}^{l+k+1} T(x_i,x_{i+1}) \wedge T(\dot{x}_{i+1},\dot{x}_{i+1}) \wedge P(x_{l+k+1},\dot{x}_{l+k+1}) \tag{14}$$

The interpolant σ contains state variables x_{l+2} and $\dot{x}_{l+2}$ of transition relation T, i.e., $\mathsf{Var}(\sigma) = \mathsf{Var}(A'') \cap \mathsf{Var}(B'')$. Intuitively, σ computes an approximated set of pairs of successor states of the correct states and faulty states reached after injecting a fault. The variables x_{l+2} and $\dot{x}_{l+2}$ of the interpolant σ are replaced by x_{l+1} and $\dot{x}_{l+1}$, respectively. The interpolant is added to X, i.e.,

$$X' = \left((\bigwedge_{\delta\in\Delta} \bar{\delta}(x_l) \wedge \mathsf{inj}(l,D)) \vee \sigma(x_{l+1},\dot{x}_{l+1}) \right) \wedge T(x_{l+1},x_{l+2}) \wedge T(\dot{x}_{l+1},\dot{x}_{l+2}) \tag{15}$$

Next, the extended formula $X' \wedge Y$ is checked for satisfiability. If the formula is satisfiable, the classification is potentially spurious. In that case, k is increased by one and the computation proceeds clearing problem instances and interpolants but keeping Δ. Otherwise, i.e., the formula is unsatisfiable, a new interpolant is computed and it is checked whether the disjunction of all previously computed interpolants implies the new interpolant – a fixed-point has been reached. This proves that no fault injection in components of D is observable at the outputs. Thus, all components of D are proven unbounded dangerous and constitute the set U.

D. Algorithm

The overall procedure which exploits Theorem 1 and the fixed-point iteration on the property from Section IV-C is

Algorithm 1: ROB-COMPL

```
   Input: $C = (V, E, L)$ a sequential circuit
   Output: $(\mathbb{S}, \mathbb{T}, \mathbb{U})$ with $\mathbb{S} \subseteq V$ the non-robust and $\mathbb{T} \subseteq V$ the robust as
           well as $\mathbb{U}$ unbounded dangerous comps.
1  begin
2      $k = l = 0; \Delta = \emptyset$;
3      $\mathbb{S} = \mathbb{T} = \mathbb{U} = \emptyset$;
4      $D = V$;
5      while true do
6          $\mathbb{S} = \mathbb{S} \cup \mathsf{sol}(\phi(l, k, D))$;
7          $D = D \setminus \mathbb{S}$;
8          if $(\mathbb{S} \cup \mathbb{T} = V)$ or $(k = k')$ then return $(\mathbb{S}, \mathbb{T}, \mathbb{U})$;
9          $\delta = \mathsf{itp}(\mathsf{inj}(l, D) \wedge \mathsf{prop}_{\mathrm{NR}}(l, k), \mathsf{reach}(l))$;
10         if $\delta$ is an adequate approximation then
11             $\Delta = \Delta \cup \delta$;
12         end
13         if necessary states are checked with respect to $k$ then
14             $l = 0$;
15         else
16             $l = l + 1$;
17             continue;
18         end
19         $\hat{\mathbb{S}}_l^k = \mathsf{sol}(\hat{\phi}(l, k, D, \Delta))$;
20         $\hat{\mathbb{D}}_l^k = \mathsf{sol}(\hat{\psi}(l, k, D \setminus \hat{\mathbb{S}}_l^k, \Delta))$;
21         $\check{\mathbb{T}}_l^k = V \setminus (\hat{\mathbb{S}}_l^k \cup \hat{\mathbb{D}}_l^k)$;
22         $D = D \setminus \check{\mathbb{T}}_l^k$;
23         $\mathbb{T} = \mathbb{T} \cup \check{\mathbb{T}}_l^k$;
24         if $\mathbb{T} \cup \mathbb{S} = V$ then
25             return $(\mathbb{S}, \mathbb{T}, \mathbb{U})$
26         else
27             if $k = 0$ then $k = 1$; continue;
28             $Q = \bigwedge_{\delta \in \Delta} \bar{\delta}(x_l) \wedge \mathsf{inj}(l, \hat{\mathbb{D}}_l^k)$;
29             $X = T(x_{l+1}, x_{l+2}) \wedge T(\dot{x}_{l+1}, \dot{x}_{l+2})$;
30             $Y = \bigwedge_{i=l+2}^{l+k+1} T(x_i, x_{i+1}) \wedge T(\dot{x}_{i+1}, \dot{x}_{i+1}) \wedge$
               $P(x_{l+k+1}, \dot{x}_{lk+1})$;
31             repeat
32                 $\sigma = \mathsf{itp}(Q \wedge X, Y)$;
33                 if $Q \rightarrow \sigma$ then
34                     $\mathbb{U} = \mathbb{U} \cup \hat{\mathbb{D}}_l^k$;
35                     $D = D \setminus \hat{\mathbb{D}}_l^k$;
36                     continue;
37                 else
38                     $Q = Q \vee \sigma(x_{l+1}, \dot{x}_{l+1})$;
39                 end
40             until $Q \wedge X \wedge Y$ is satisfiable ;
41             $k = k + 1$;
42         end
43     end
44 end
```

shown as pseudo-code in Algorithm 1. The algorithm gets the circuit under verification as input and determines the set of non-robust as well as robust components. Lines 2-4 initialize the required sets and the values for l and k. All components are to be classified, i.e., $D = V$. The `while`-loop iterates until all components are classified or the threshold for k is reached (line 8). In each iteration at first non-robust components are determined (line 6) and are excluded from the components to be classified in further iterations (line 7). Next, in line 9 an interpolant is computed based on Formula (10) and it is checked whether the interpolant is an adequate approximation using interpolation-based model checking (line 10). If the computed interpolant is an adequate approximation, then the interpolant is added to the set Δ (line 11). If all necessary states are covered as checked by interpolation-based fixed-point computation, then l is set to zero and the algorithm proceeds with line 19. Otherwise, the value of l is increased by one (line 16) if not all necessary states are checked for the current value of k and the outer loop restarts. That means either further classifications are performed or an adequate approximation will be found by strengthening A' by increasing l by one. Eventually an adequate approximation will be found according to Theorem 1 and the procedure continues with line 19. Here, the over-approximations of the set of non-robust and dangerous components are computed, such that a subset of the robust components is determined. In line 27–41 the fixed-point computation on the property is performed. If a fixed-point is found, all components previously classified as dangerous in line 22 are proven to be unbounded dangerous components. If the formula becomes satisfiable, k is increased by one to get a weaker approximation by interpolation on the property. In a next iteration further non-robust components may be classified from the set of remaining dangerous components.

E. Adequate Approximation in Model Checking

Since, the computed adequate approximations are invariants specifying reachable states of the circuits they can be used in model checking in general. While model checking to prove a property, adequate approximations can be derived in the same way as described above. These adequate approximations may be applied as invariants to prune the search space while proving other properties.

V. Experiments

An evaluation of the proposed approaches is presented in this section. Experiments have been carried out on a Dual-Core AMD Opteron$^{\text{TM}}$ Processor with 3.0 Ghz and 64GB main memory under Linux. Reaching a timeout of 15 hours is marked by *timeout*. As SAT-solver MiniSAT's proof logging version [20] has been used and interpolants are computed based on McMillan's interpolation system [10]. Preliminary experiments have shown that the HKP [17], [18], [19] interpolants yielded consistently longer run times of the proposed algorithms than using McMillan's interpolants. Interpolants are represented as *And-Inverter-Graphs* (AIG) by the Aiger software package[1].

A. ITC'99 Benchmarks

For the first evaluations ITC'99 benchmarks were taken and enhanced with TMR techniques to catch single transient faults. TMR circuits are marked with the suffix `-tmr`. Faults are injected into gates and flipflops, i.e., initially $D = V$. The two proposed approaches from Section III and Section IV as well as the approach from the work of [9] are compared in this section.

The circuits are known to be hard for formal robustness checking, because fault effects propagate until reaching the

[1]Available under: http://fmv.jku.at/aiger/

				APPROACH OF [9]			INTERP.-BASED BMC			COMPLETE APPROACH							
Circuit	|IN|	|OUT|	|FF|	R_{lb}	R_{ub}	Runtime	k	R_{ub}	Runtime	l	k	invalid	valid	∅ size	R_{lb}	R_{ub}	Runtime
b01-tmr	2	2	15	7.9	98.2	*timeout*	236	97.8	*timeout*	17	4	15	4	3k	97.8	97.8	3
b02-tmr	1	1	12	1.7	98.2	*timeout*	351	98.0	*timeout*	25	9	23	4	1k	98.0	98.0	<1
b03-tmr	4	4	90	1.2	98.7	*timeout*	40	98.7	*timeout*	24	4	22	4	91k	98.7	98.7	31
b04-tmr	11	8	198	0.6	100.0	*timeout*	1	100.0	*timeout*	6	1	4	1	3583k	0.0	99.2	*timeout*
b05-tmr	1	36	102	6.9	99.0	*timeout*	70	98.9	*timeout*	16	4	14	3	29k	98.9	98.9	319
b06-tmr	2	6	27	3.6	97.0	*timeout*	28	96.7	*timeout*	9	3	7	3	6k	96.7	96.7	11
b07-tmr	1	8	147	0.9	99.5	*timeout*	72	99.4	*timeout*	21	4	19	3	22k	99.4	99.4	1621
b08-tmr	9	4	63	1.2	99.4	*timeout*	25	99.4	*timeout*	14	5	12	3	4k	99.4	99.4	68
b09-tmr	1	1	84	0.3	99.6	*timeout*	84	99.6	*timeout*	19	4	17	4	6k	99.6	99.6	44
b10-tmr	11	6	51	1.5	97.8	*timeout*	9	97.8	*timeout*	16	4	14	3	1286k	97.8	97.8	778
b11-tmr	7	6	93	0.6	99.4	*timeout*	11	99.4	*timeout*	13	2	11	3	537k	99.4	99.4	373
b12-tmr	5	6	363	0.3	99.8	*timeout*	12	99.8	*timeout*	19	2	17	3	1012k	99.8	99.8	395
b13-tmr	10	10	159	2.3	99.0	*timeout*	4	99.0	*timeout*	7	3	5	3	9k	99.0	99.0	213

TABLE I: Determined robustness of ITC'99 circuits

majority voter and are then masked late in the design. This structure is hard for most SAT-solvers to handle causing long run times. Furthermore, faults on most components do not affect the primary outputs of the circuit, i.e., the faults are masked by the majority voter, but modify the state. That means, in each iteration most components are reconsidered as dangerous until they are finally proven to be unbounded dangerous components. The approach of [9] and from Section III are practically not able to compute the unbounded dangerous components, due to the large value of k'. The new complete approach that does not explicitly unroll the transition relation for k' time frames.

Results are shown in Table I. Properties of the circuit like name, number of primary inputs, number of primary outputs and number of state elements are shown in the first four columns. Values for the robustness are computed as follows:

$$R_{lb} = \frac{|\mathbb{T} \cup \mathbb{U}|}{|V|}, \qquad R_{ub} = 1 - \frac{|\mathbb{S}|}{|V|}. \tag{16}$$

The runtimes are given in seconds.

Once the computation is finished by the evaluated approaches, the robustness is computed based on the respective sets of non-robust and robust components returned by the approaches. The approach from Section III computes only a set of non-robust components and therefore only the upper bound of the robustness R_{ub} is computed. Furthermore, the values for l and k are shown for the complete approach in Table I, to denote when the approach finished with a complete answer. Furthermore, details about the interpolants are given. The approach computes interpolants to get adequate approximations. Column *invalid* denotes the number of interpolants which are not an adequate approximation. Column *valid* denotes the number of adequate approximations. Column *∅ size* denotes the average size of the interpolants given as number of nodes of the AIGs. The run times to check whether an interpolant is an adequate approximation are quite short, i.e., less than 5 minutes accumulated over all instances.

For the approach from [9] the gap of the bounds is very large for each considered circuit. This happens, because almost all faults are masked by the majority voter of the TMR circuit, but are not corrected such that one of the TMR modules is corrupted unless resetting the circuit. That means, the approach of [9] was not able to prove that the dangerous components will not affect the primary outputs at any time.

The approach from Section III computes a tight upper bound. As shown in the table the approach classifies some more components as non-robust than the approach of [9] for the circuits, `b01-tmr`, `b02-tmr`, `b04.tmr`, `b05-tmr`, `b06-tmr`, `b07-tmr`. Rather than a pre-defined under-approximation of reachable states [9], necessary reachability information is computed.

In principle, the approach of [9] uses fixed values $l = 10$ and $k = 10$ that prevent a complete classification. The BMC-based approach does not terminate even for a large observation window as no fixed-point iteration after fault injection is performed. However, a complete classification is possible even for a small observation window k which also requires precise reachability information. This shows that both interpolation steps are required for an effective classification.

The complete approach proves lower and upper bounds very effectively and provides an exact analysis for almost all circuits. In these cases lower bound and upper bound meet.

B. IBM Benchmarks

Next, we checked the proposed complete method on a set of IBM design blocks. In this setting, two groups of design blocks were used: 1) blocks taken from a data-path design, and 2) blocks taken from a micro-processor control unit. All of these design blocks contain self error checking and correction mechanisms implemented in hardware. In this benchmark, an Intel i5 processor running at 3.1GHz with 4GB RAM was used. Table II summarizes the benchmark results: blocks D1-13 are the data-path blocks, D14-D27

Circuit	|IN|	|OUT|	|FF|	Classified [%]	l	k	Runtime
D1	204	259	1430	9.30%	2	0	2728
D2	228	65	1424	17.49%	5	1	783
D3	727	293	1395	7.74%	3	0	220
D4	700	497	1038	70.52%	7	1	678
D5	364	142	940	100.00%	2	1	60
D6	105	60	699	99.86%	2	57	1699
D7	284	262	513	84.99%	18	3	3797
D8	112	56	456	100.00%	6	2	144
D9	268	99	447	89.26%	8	1	8281
D10	734	194	435	87.36%	2	1	611
D11	155	120	394	100.00%	2	1	11
D12	53	37	322	100.00%	2	1	19
D13	124	67	222	48.20%	5	3	37
D14	119	112	878	81.55%	5	3	1492
D15	140	55	804	88.56%	31	0	710
D16	29	24	555	15.86%	61	1	1201
D17	377	25	506	70.16%	6	6	1504
D18	176	154	464	70.26%	55	1	2044
D19	252	131	451	56.54%	7	7	1714
D20	327	102	428	67.06%	3	44	9050
D21	173	256	412	88.35%	8	4	486
D22	135	206	247	90.28%	2	33	23170
D23	218	96	231	95.24%	2	86	3631
D24	119	57	231	96.54%	2	2	580
D25	227	114	216	95.37%	4	95	7589
D26	70	51	210	17.62%	131	0	3697
D27	103	63	207	91.30%	7	6	598
D28	130	79	195	95.90%	2	80	35888
D29	100	37	126	100.00%	5	5	353
D30	139	94	123	100.00%	4	5	59

TABLE II: IBM Benchmarks

are the control logic blocks, number of flip flops, primary inputs and primary outputs are given for each design in the |IN|, |OUT| and |FF| columns respectively. In addition, the percentage of classified flip-flops are given, as well as the l and k bounds which were reached during each run.

One can see in Table II, the method was able to classify significant percentage of flip-flops in most of the blocks. In case where the implementation ran out of memory less than 100% of the flip flops were classified. This can be attributed to several reasons: 1) The underlying SAT solver simply ran out of memory trying to solve a particular instance, 2) problem instance itself may get too large to fit into memory. In addition it is worth noting that in the current implementation the interpolants are not being optimized, which might also contribute to sub-optimal performance. However, the implementation of the newly introduced approach was able to effectively classify industrial design coming from a micro-processor design.

VI. Conclusion

This work proposed a new complete approach for robustness checking utilizing interpolants to overcome complexity problems. We over-approximate the exact set of reachable states and compute a fixed-point on the property. Our approach is effective and provides exact results for hard benchmarks.

References

[1] S. Krishnaswamy, S. M. Plaza, I. L. Markov, and J. P. Hayes, "Enhancing design robustness with reliability-aware resynthesis and logic simulation," in *ICCAD*, 2007, pp. 149–154.

[2] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-based attack of RSA authentication," in *DATE*, 2010, pp. 855–860.

[3] J. Hayes, I. Polian, and B. Becker, "An analysis framework for transient-error tolerance," in *VTS*, may 2007, pp. 249–255.

[4] N. Miskov-Zivanov and D. Marculescu, "Multiple transient faults in combinational and sequential circuits: A systematic approach," *IEEE Trans. on CAD*, vol. 29, no. 10, pp. 1614 –1627, 2010.

[5] M. Bozzano, A. Cimatti, and F. Tapparo, "Symbolic fault tree analysis for reactive systems," in *ATVA*, ser. LNCS, vol. 4762, 2007, pp. 162–176.

[6] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *IOLTS*, 2005, pp. 260–265.

[7] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *DATE*, 2007, pp. 1442–1447.

[8] M. Hunger, S. Hellebrand, A. Czutro, I. Polian, and B. Becker, "ATPG-based grading of strong fault-secureness," in *IOLTS*, 2009, pp. 269 –274.

[9] G. Fey, A. Sülflow, S. Frehse, and R. Drechsler, "Effective robustness analysis using bounded model checking techniques," *IEEE Trans. on CAD*, vol. 30, no. 8, pp. 1239 –1252, 2011.

[10] K. L. McMillan, "Interpolation and SAT-Based model checking," in *CAV*, ser. LNCS, 2003, pp. 1–13.

[11] J. Marques-Silva, "Interpolant learning and reuse in SAT-based model checking," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 3, pp. 31–43, May 2007.

[12] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *VMCAI*, ser. LNCS, vol. 5944, 2010, pp. 129–145.

[13] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Boosting interpolation with dynamic localized abstraction and redundancy removal," *TODAES*, vol. 13, no. 1, pp. 1–20, 2008.

[14] V. D'Silva, M. Purandare, and D. Kroening, "Approximation refinement for interpolation-based model checking." in *VMCAI*, ser. LNCS, 2008, pp. 68–82.

[15] S. F. Rollini, O. Sery, and N. Sharygina, "Leveraging interpolant strength in model checking," in *CAV*, Springer. Berkeley, California, USA: Springer, 2012.

[16] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.

[17] G. Huang, "Constructing Craig interpolation formulas," in *Annual International Conference on Computing and Combinatorics*, 1995, pp. 181–190.

[18] J. Krajicek, "Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic," *The Journal of Symbolic Logic*, vol. 62, no. 2, pp. 457–486, 1997.

[19] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *The Journal of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.

[20] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, 2003, pp. 502–518.

Symbolically Synthesizing Small Circuits

Rüdiger Ehlers[1], Robert Könighofer[2], and Georg Hofferek[2]

[1]Reactive Systems Group, Saarland University, Germany
[2]Institute for Applied Information Processing and Communications, Graz University of Technology, Austria

***Abstract*—*Reactive synthesis*, where a finite-state system is automatically generated from its specification, is a particularly ambitious way to engineer correct-by-construction systems. In this paper, we propose *implementation-extraction based on computational learning of Boolean functions* as a final synthesis step in order to obtain *small and fast circuits* for realizable specifications in a symbolic way. Our starting point is a restriction of the system player's choices in a synthesis game such that all remaining strategies are winning. Such games are used in most symbolic synthesis tools, and hence, our technique is not tied to one specific synthesis workflow, but rather supports a large variety of these. We present several variants of our implementation-learning approach, including one based on Bshouty's monotone theory. The key idea is the efficient use of the system player's freedom in the game. Our experimental results show a significant reduction of implementation size compared to previous methods, while maintaining reasonable computation times.**

I. Introduction

A common criticism on formal methods for the verification of reactive systems is that they only aid the system engineer with ensuring correctness *after* the system is constructed. The idea of *reactive synthesis* is to change this situation by automatically computing a correct-by-construction system after the specification is stated. While the theoretical complexity of the synthesis problem is long-established for many important specification formalisms, only recently, progress on the practical solution of this problem has gained momentum, as new results on symbolic synthesis [18], [17], smart specification decomposition techniques [16], [29], and specification formalisms explicitly targeting synthesis have emerged [7].

Early theory on the subject was mainly concerned with checking the *realizability* of a specification, i.e., testing if there exists an implementation. Procedures for obtaining an implementation in case of a positive answer were rudimentary and did not consider the *quality* of the synthesized solutions. With the rise of synthesis technology from its infancy, a growing interest in synthesizing solutions of high quality (e.g., requiring only a small on-chip area, or reacting quickly) emerged, witnessed by the introduction of synthesis methods that take quantitative criteria into account. Introducing quantitative criteria at the start of the synthesis process however typically breaks the possibility to perform the synthesis process in a symbolic way, e.g., using binary decision diagrams (BDDs).

This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), by the German Science Foundation (DFG) through the AVACS Transregional Collaborative Research Center (SFB/TR 14), and by the Austrian Science Fund (FWF) through project QUAINT (I774-N23) and the national research network RiSE (S11406-N23).

The key idea to solve this problem is to consider the quality only at a later stage in the synthesis process, when the realizability of a specification has already been determined. Most synthesis approaches reduce the realizability problem of a given specification to solving a two-player game in which one player models the environment and provides the input to a system to be synthesized, whereas the other player models the system and provides the output. If and only if the system player can always win the game, the specification is realizable. We call the characterization of a set of moves a *general strategy* if the system player wins when taking only moves from this set. Calling it *general* is justified by the fact that there are often situations in which the strategy can be *non-deterministic*, i.e., it has multiple choices for the system player to win in that situation. Any implementation that resolves this non-determinism, and is thus a *specialization* of such a general strategy, satisfies the original specification. Since a general strategy is a natural by-product of most game solving algorithms used in synthesis, we can easily take the general strategy as input to a process for finding a small implementation. This way, we have separated the problems of synthesizing *any* solution and obtaining a *good* one. While we might miss the smallest implementation this way, we do not introduce any additional computational hassle by combining the two goals in one step.

Theoretical work on computing small implementations from general strategies shows that approximating the size of a smallest finite-state machine within any polynomial quality function [14] from a general strategy is NP-hard.[1] This holds even if we do not have any input to the system. The fact that for scalable synthesis, we also have to be able to cope with symbolically represented general strategies[2], and also want a circuit rather than an explicit finite-state machine as result, does not quite make the problem easier in terms of complexity. As a consequence, current methods minimize the circuit size heuristically while only guaranteeing correctness. Experience however shows that with these techniques, the circuit sizes are often prohibitively large, calling for a better approach.

In this paper, we present a learning-based approach to com-

[1]For example, any algorithm that (1) outputs **false** if for some given general strategy and value of n, there exists no finite-state machine of size n that behaves in a way allowed by the general strategy, (2) outputs **true** if there exists such a finite-state machine of size at most n^{10}, and (3) outputs an arbitrarily result otherwise, solves an NP-hard problem.

[2]A symbolic representation of the synthesis game and the general strategy is crucial for scalability of reactive synthesis because the transformation of a specification into a game can lead to huge state spaces.

puting small circuits in symbolic reactive synthesis. In contrast to previous approaches, we do not exploit special properties of the data structure involved for symbolic reasoning (such as BDDs), but rather use computational learning of Boolean functions. This allows us to utilize the *non-determinism* of general strategies in a much more effective way. We learn a CNF (conjunctive normal form), DNF (disjunctive normal form), CDNF (conjunction of DNFs), or DCNF (disjunction of CNFs) representation of output and next-state bit valuations, which can immediately be translated into circuits. These circuits are not only typically smaller, but also more shallow than those of previous approaches, which allows running them at higher clock rates.

Our approach is not bound to one specific synthesis workflow, but supports any flow that computes a general strategy. For our experimental evaluation, we used two different BDD-based synthesis tools, namely RATSY [4], and UNBEAST [15]. RATSY provides us with general strategies stemming from the generalized reactivity(1) synthesis approach [7]. UNBEAST is a symbolic implementation [16] of a bounded synthesis variant [18]. In our experiments, we obtained circuit-size improvements of around one order of magnitude, when compared to the built-in approaches of these tools. The computation times are longer but still reasonable, thus allowing the new approach to be applied also to large problem instances.

This paper is structured as follows. In the next section, we give an overview of related work and provide experiences with previous approaches to circuit computation in synthesis. Then, we briefly discuss preliminaries and give literature pointers to the computation of general strategies in synthesis workflows. In Section IV, we describe our new learning-based approach, followed by an experimental evaluation in Section V. We conclude with a summary and ideas for future work.

II. Previous Approaches and Related Work

Computing an implementation in case of a realizable specification is the last step of every reactive synthesis approach. There are a few of these for which this last step is an easy one. In SMT-based bounded synthesis [18], the realizability of a specification by some finite-state machine with b states is encoded into a satisfiability modulo theory (SMT) formula, whose solution is an explicit implementation. Anti-chains-based bounded synthesis [17] uses anti-chains, rather than BDDs, as symbolic data structure during a game-solving process. It is then trivial to extract an implementation with as many states as there are elements in the final anti-chain.

Both approaches come at a price. SMT-based bounded synthesis is only reasonable if there exist small implementations and the number of input/output signals is not too high. Anti-chains-based bounded synthesis requires the specification to be a conjunction of relatively small sub-specifications in order to scale well. To counter these limitations, we are concerned with general circuit extraction approaches that start with symbolic general strategies. In the remainder of this section, we describe previous techniques for this task, state our experiences with them, and discuss techniques similar to our new approach.

Kukula and Shiple [23] described a simple technique to compute a circuit from a general strategy in BDD form. The main idea is to take the graph structure of the BDD and instantiate an 8-gate building block for all nodes to obtain an implementation. The resulting circuits have a very high depth (more than two times the number of state and input variables) and experience shows that they are often huge [6].

ANZU [21] uses a simple, cofactor-based approach [6] to compute a completely specified Boolean function for each output signal. The BDDs that represent these functions are then dumped into a network of multiplexers. Bloem et al. [6] also mention a simple but effective optimization. For each output, they remove unnecessary input variables by existential quantification. This method has also been implemented in RATSY [4]. To the best of our knowledge, this is the most effective circuit synthesis approach previously known, and it will be used as a baseline for comparison in Section V. We will subsequently refer to this method as the *cofactor approach*.

Baneres et al. [3] present a recursive paradigm for extracting completely specified Boolean functions from general strategies. Their approach is based on first computing the single output functions independently, without resubstitution. In a second stage they recursively resolve inconsistencies resulting from uncoordinated choices during the first stage. They also introduce a recursion-depth limit. If the limit is reached, their algorithm falls back to an arbitrary other relation-solving method. We reimplemented their approach within RATSY and applied it to its general strategies. Unfortunately, first experimental results were rather discouraging. Without any recursion limit, the approach timed out even for rather small benchmarks. However, using a recursion limit, we (almost) always hit the fall-back mechanism. The result of the fall-back mechanism is in almost all cases the same as if the recursive approach of [3] had not been used at all. Therefore, this approach does not provide any improvement concerning circuit size, but only increases computation time significantly. We believe that this is due to the fact that our general strategies are highly non-deterministic, and in particular have many vertices that [3] calls “non-don’t-care extendable”.

Another approach that we tried was implementing the Minato-Morreale algorithm for computing an *Irredundant Sum-of-Products* [24], [26]. It is a recursive procedure that takes a general strategy as an input and computes a Sum-of-Products form for a compatible completely specified function. The final result is *irredundant* in the sense that no single literal or cube can be deleted without changing the function. We use the recursive structure of the algorithm to build a multi-level Boolean circuit along the way. The resulting circuits are comparable in size to the ones obtained through the cofactor approach. Computation times, however, are significantly higher. To further improve these results, we also tried using a “cache”. In each step, the algorithm first checks whether a function lying in the desired interval of functions has already been built as a circuit in previous steps. If so, this function (and the corresponding wire in the circuit) is reused. To keep the memory footprint of the cache small and to speed up the process of a cache look-up, we did not store the BDDs of the functions, but rather used a signature-based approach as in [25]. We only store the function's output for some random input vectors. These outputs are called a *signature*. Signatures have a very low memory footprint. When doing a look-up, we can use the signature to perform a fast pre-test. This pre-test may, however, create false positives. Thus, whenever the pre-test yields a positive result, we (recursively) reconstruct a BDD for the function in question from the structure of the circuit generated so far. We subsequently use this BDD to perform a sound comparison to check whether or not the function really lies within the desired interval. Experimental results have shown that, unfortunately, we get almost no cache hits. The hits we do get are mostly very small, almost trivial functions, consisting of only a handful of gates. Thus, the gain due to sharing is negligible. On the other hand, computation time rises significantly due to the many look-up checks that have to be performed. We also noticed that, when extended from completely specified functions [25] to intervals, the signature-based pre-test gives too many false positives to be of use.

Jiang et al. [19] presented a SAT-solver-based approach to compute functions from a general strategy. Their method is based on Craig interpolation [13], which is supported by many modern SAT solvers. Also here, preliminary experimental results suggest that this method cannot deal well with the high degree of non-determinism which is characteristic for general strategies in reactive synthesis. First tests produced circuits that were at least one order of magnitude larger than the ones obtained by the cofactor approach.

Our method for computing circuits is based on computational learning. It starts with simple candidate functions and refines them based on the counterexamples that are returned by a teacher oracle. Counterexample-guided refinements have already been used in program sketching [30] to synthesize missing program parts, and for program repair [22], [10]. Natively, these methods can only synthesize integer constants. Templates or user-provided generators containing unknown integers are used to synthesize more sophisticated program parts. In contrast, our method is able to compute circuits directly and without the help of the user.

Computational learning of Boolean function has many applications apart from implementation extraction. Becker et al. [2] use the concept to turn a quantified Boolean formula (QBF) solver into a tool for obtaining a compact representation of *all* solutions to a Boolean formula that may or may not have some quantifiers. While the representation types for the solutions are the same as in this work (CNF, DNF, and a conjunction of DNF formulas), Becker et al. focus on integrating a QBF solver into the classical learning algorithms for these representations. In this paper, on the other hand, we are not concerned with such low-level technical considerations, and simplify the details of our approach by taking BDDs as data structure for symbolic reasoning. This allows us to start right away with tackling the special properties of the implementation extraction domain, in particular how to obtain efficient circuits with multiple output signals, and how to make the best use of the non-determinism in the general strategies.

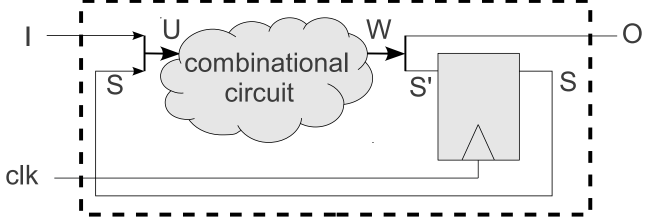


Fig. 1. Implementation of a general strategy.

The work in [11] addresses learning of Boolean functions over enlarging sets of variables, especially for loop invariant generation and assumption synthesis. The learning method is based on Bshouty's monotone theory [8], just like one of our algorithms. However, while [11] concentrates on efficiency in presence of an unbounded number of variables, we focus on utilizing non-determinism effectively to obtain small circuits.

III. Preliminaries

A. Basic Notation

Let V be a set of Boolean variables. To simplify notation, we treat subsets X of V and their characteristic functions interchangeably. Thus, a subset X of V induces a variable valuation $x : V \to \mathbb{B}$ by setting $x(v) = \mathbf{true}$ for some $v \in V$ if and only if $v \in X$, and likewise, a variable valuation $x : V \to \mathbb{B}$ induces a subset X of V by choosing $X = \{v \in V \mid x(v) = \mathbf{true}\}$. A model of a Boolean formula is a variable valuation that satisfies the Boolean formula.

B. General strategies

A *general strategy* is a tuple $\mathcal{S} = (S, I, O, s_0, \delta)$, where S is a set of *state bits*, I is a set of *input bits*, O is a set of *output bits*, $s_0 \subseteq S$ is the *initial state* and $\delta \subseteq 2^S \times 2^I \times 2^S \times 2^O$ is the *transition relation*. To separate the two occurrences of state bit sets in the transition relation, we will henceforth write S' for their second copy. In this paper, we are concerned with *extracting* a small circuit for this strategy, i.e., a net with input bits $U = S \cup I$ and output bits $W = S' \cup O$ such that for every input $(s, i) \in 2^S \times 2^I$, if s is a *reachable state* and the circuit outputs some $(s', o) \in 2^{S'} \times 2^O$ for this input, then $(s, i, s', o) \in \delta$. We call such a circuit a *specialization* of $\mathcal{S}$ as it exhibits only behavior that is allowed by the strategy, and chooses one particular output/next state combination whenever more than one is possible. A state s is considered to be reachable if there exists some sequence $s_0 \xrightarrow[i_0]{o_0} s_1 \xrightarrow[i_1]{o_1} \ldots \xrightarrow[i_n]{o_n} s_n$ with $s_n = s$ such that for all $j \in \{0, \ldots, n\}$, (s_{j+1}, o_j) is the output of the circuit for the input bit valuation (s_j, i_j). Thus, we assume that some flip-flops feed back the output state bits as input to the net in the next computation cycle. This implementation of $\mathcal{S}$ in a circuit is illustrated in Fig. 1. For the scope of this paper, the sizes and depths of combinatorial circuits are considered at the gate level.

General strategies as defined above are computed in many modern synthesis workflows as a by-product. Normally, not all possible implementations of a specification are specializations of the general strategy computed, but rather some unfavorable ones have already been filtered out. For example, if we used a mutual-exclusion protocol specification as input to a synthesis tool, we would want that the general strategy computed ensures that grants are given quickly in order not to let the requester wait unnecessarily. For the synthesis approaches considered in this paper, this *good reactivity* is ensured: in generalized reactivity(1) synthesis [7], [27], the general strategy is built such that transitions that help towards the fulfillment of liveness objectives are preferred, whereas in BDD-based bounded synthesis [16], a strict bound on delays in such a situation is imposed. Thus, in both cases, for synthesizing implementations of high quality, we can restrict our attention to circuit size and depth for the scope of this paper. For details on the computation of general strategies in these two synthesis approaches, the interested reader is referred to [16], [7], [27].

C. Binary Decision Diagrams

To handle Boolean functions and general strategies symbolically, we use *reduced ordered binary decision diagrams* (BDDs) [9], which represent characteristic functions $f : 2^V \to \mathbb{B}$ for some finite set of variables V. Let f and f' be two BDDs and $V' \subseteq V$ be a set of variables. We denote the conjunction, disjunction, negation, existential quantification, and universal quantification of BDDs as $f \wedge f'$, $f \vee f'$, $\neg f$, $\exists V' . f$ and $\forall V' . f$, respectively. To represent the transition relation of a general strategy $\mathcal{S} = (S, I, O, s_0, \delta)$, we take $V = S \uplus I \uplus S' \uplus O$ and build the characteristic function of δ by setting $f_\delta(X) = \mathbf{true}$ for some $X \subseteq V$ iff $(X \cap S, X \cap I, X \cap S', X \cap O) \in \delta$.

IV. Learning Small Circuits

In this section, we present the core contribution of this work: Given a general strategy $\mathcal{S} = (S, I, O, s_0, \delta)$, we show how to compute a combinatorial circuit with input bits $U = S \cup I$ and output bits $W = S' \cup O$ that implements the strategy as illustrated in Fig. 1. We break down this problem into obtaining $|W|$ circuits with a single output bit, each getting $|U|$ bits as input. First, we describe this decomposition process. Then, in Section IV-B, we discuss how a circuit for a single output bit can be computed using computational learning.

A. Decomposition

One reason for computing $|W|$ one-output-bit circuits, as we do in this paper, is to increase the freedom in the circuits due to unreachable states. If a circuit for one output bit has been found, we can recompute the set of states reachable by any specialization that uses this circuit. Typically, this reachable state set shrinks with every additional circuit, which allows our specialization to ignore more and more input valuations $X \subseteq U$ as the algorithm proceeds. The following algorithm describes the overall process:

```
1: procedure ObtainCircuit($S, I, O, s_0, \delta$)
2:    $A := \delta$
3:    for $v \in W$ do
4:       $r :=$ states reachable from $s_0$ under $A$ as BDD
5:       $c := r \wedge (\neg(\exists W . (A \wedge v)) \vee \neg(\exists W . (A \wedge \neg v)))$
6:       if optimize then
7:          $(f, c) :=$ MinVars($A, v, c$)
8:       else
9:          $f := \exists W . (A \wedge v)$
10:      end if
11:      $g :=$ Learn($U, f, c$)
12:      Take $g$ as output circuit for $v$
13:      $A := A \wedge (\neg v \oplus g)$
14:   end for
15: end procedure
```

The algorithm iterates over all outputs $v \in W$ of the combinatorial circuit we wish to build. In every iteration, we first compute which states are reachable in any specialization with the circuits computed so far. Next, we compute the care set, i.e., the set of input variable valuations $X \subseteq U$ for which the output matters. There are two reasons why a valuation might not be in the care set: (1) the state is not reachable, and (2) both values of the output bit v are allowed. The computation of c in line 5 reflects these cases. Ignore the optional optimization in line 7 for a moment. Line 9 now computes the target function, and line 11 uses a black-box function Learn to obtain a corresponding circuit using computational learning. We assume that Learn returns a BDD representation of the one-output-bit circuit that resembles f on variable valuations $X \subseteq V$ for which $c(X) = \mathbf{true}$ holds. We describe two variants of a function Learn in the next subsection.

After a circuit for one output $v \in W$ has been obtained, we need to update the general strategy to only allow output variable valuations $X \subseteq W$ that are still possible when using the circuits we already have. This happens in line 13.

The optimization in line 7 minimizes the number of input bits $v' \in U$ on which the output bit v may depend. This idea has been introduced in [6] and can be implemented as follows.

```
1: procedure MinVars($A, v, c$)
2:    $m_0 := \neg((\exists W . (A \wedge v)) \vee \neg c)$
3:    $m_1 := \neg((\exists W . (A \wedge \neg v)) \vee \neg c)$
4:    for $v' \in U$ do
5:       $(m_0', m_1') := (\exists v' . m_0, \exists v' . m_1)$
6:       if $m_0' \wedge m_1' = \mathbf{false}$ then
7:          $(m_0, m_1) := (m_0', m_1')$
8:       end if
9:    end for
10:   return $(m_1, m_0 \vee m_1)$
11: end procedure
```

The lines 2 and 3 compute the input variable valuations for which the circuit to be learned has to output $\mathbf{false}$ and $\mathbf{true}$, respectively. If an existential quantification of an input variable v' does not make the two regions m_0 and m_1 overlap, then this means that the function can still be implemented without taking the input bit v' into account. Otherwise, v' is crucial for distinguishing input variable valuations for which the circuit has to output $\mathbf{true}$ from those where it has to output $\mathbf{false}$. In this case, v' cannot be disregarded. The check is done in line 6. After all unnecessary input bits have been discarded using existential quantification, the target function f and the care set c are recomputed and returned. We also use the above algorithm in a heuristic to find a good ordering for the output bits in line 3 of OBTAINCIRCUIT: output bits v for which f depends on fewer variables are considered simpler to handle, and are thus processed first.

B. Learning circuits with a single output bit

When computing a small one-output-bit circuit from a problem instance (V, f, c), our aim is to utilize *don't care* input bit valuations (i.e., $X \subseteq V$ with $c(X) = \mathbf{false}$) as effectively as possible, while obtaining circuits with appealing properties, such as low depth and few gates. We describe here how to apply the concept of *computational learning* to obtain small and shallow circuits. We decompose f into a Boolean formula that only needs to be correct on the *care set*, i.e., the input bit valuations that c maps to $\mathbf{true}$. The Boolean formula is built in an incremental fashion, i.e., we start with a small formula that we iteratively refine until it is correct with respect to the care set. After learning the formula, it can easily be translated to a circuit by using only AND, OR, and NOT gates.

We describe two variants of the learning process here, one for which the target Boolean formula is in CNF form, and one for which it is in CDNF form, i.e., a conjunction of disjunctive normal form Boolean formulas. Both variants are instances of Angluin-style [1] learning algorithms, in which the learning process proceeds by performing *queries* of various types to some teacher oracle. In our context, queries reduce to operations on BDDs. We can use the CNF and CDNF algorithms to also learn DNF and DCNF formulas: we simply dualize f and the output formula.

1) Learning CNFs: A formula in conjunctive normal form is a conjunction of clauses, each being a disjunction of literals. Given a one-bit output circuit problem (V, f, c), a clause C is *sound* in a CNF for (V, f, c) iff $\neg C(X)$ implies $\neg f(X) \vee \neg c(X)$ for all $X \subseteq V$. That is, a sound clause only evaluates to **false** if the variable valuation can be mapped to **false**. On the other hand, if a CNF formula of sound clauses maps every $X \subseteq V$ with $c(X) \wedge \neg f(X)$ to **false**, then it has enough clauses to be a valid solution. Using BDDs, we can easily check if a CNF formula has enough clauses or if a clause is sound. The following algorithm iteratively searches for sound clauses until we have enough of them:

```
1:  procedure LEARNCNF($V, f, c$)
2:    $r := \mathbf{true}$
3:    while $r \wedge (\neg f) \wedge c \neq \mathbf{false}$ do
4:      $b :=$ pick some variable valuation in $r \wedge (\neg f) \wedge c$
5:      $V' := V$
6:      $C := \bigvee_{v' \in V'} v' \oplus b(v')$
7:      for $v \in V$ do
8:        $C' := \bigvee_{v' \in (V' \setminus \{v\})} v' \oplus b(v')$
9:        if $((\neg C') \wedge c \wedge f) = \mathbf{false}$ then
10:         $(C, V') := (C', V' \setminus \{v\})$
11:       end if
12:     end for
13:     $r := r \wedge C$
14:   end while
15:   return $r$
16: end procedure
```

The variable r stores the candidate CNF formula as BDD. In practice, we store the BDD together with the corresponding CNF formula to avoid reconstructing the CNF at the end. In line 3, we check if we have found enough sound clauses already. If this is not the case, we pick some variable valuation b that witnesses this fact. We use b to derive a new sound clause in line 6. In the lines 7 to 12 we shorten it as much as possible while retaining its soundness. This way, we keep both the length and number of the clauses in the formula small.

Let $\mathcal{X} = \{X \subseteq V \mid c(X) \wedge \neg f(X) \wedge r(X)\}$ be the set of variable valuations that r must, be but does not yet, map to **false**. LEARNCNF terminates because it quits on $|\mathcal{X}| = 0$, and $|\mathcal{X}|$ decreases in every loop iteration. It is correct because it adds only sound clauses to r, and enough to have $|\mathcal{X}| = 0$.

After LEARNCNF is finished, we remove all clauses from r for which removing leaves the learned function consistent with f on c. This reduces the size of the learned function.

Example 1. *We illustrate* LEARNCNF *on the learning problem* $(\{v_1, v_2\}, f, c)$*, defined in the left part of the following truth table.*

v_1	v_2	f	c	r^1	$C^1 = \neg v_1 \vee v_2$	$C'^1 = r^2 = v_2$
false	**false**	**false**	**false**	**true**	**true**	**false**
false	**true**	**false**	**false**	**true**	**true**	**true**
true	**false**	**false**	**true**	**true**	**false**	**false**
true	**true**	**true**	**true**	**true**	**true**	**true**

We write a^i to denote variable a in iteration i of LEARNCNF*. The first candidate is $r^1 = \mathbf{true}$. An input valuation b^1 satisfying $r_1 \wedge (\neg f) \wedge c$ is $b^1(v_1) = \mathbf{true}$, $b^1(v_2) = \mathbf{false}$; the corresponding clause is $C^1 = \neg v_1 \vee v_2$. Next, C^1 is simplified by removing literals as long as soundness is preserved. $C'^1 = v_2$ renders $(\neg C'^1) \wedge c \wedge f$ unsatisfiable, so C'^1 is still sound. Since the empty clause is not sound, r is refined to $r^2 = C'^1 = v_2$. Now, $r_2 \wedge (\neg f) \wedge c$ is unsatisfiable, i.e, there is no more input valuation for which the circuit must, but does not yet, output* **false**. *Hence, the circuit outputting $r_2 = v_2$ is a solution to the learning problem $(\{v_1, v_2\}, f, c)$.*

2) Learning CDNFs: The CNF learning approach above computes two-level combinatorial circuits. While these are shallow, there are many functions for which we need more levels in order to obtain a circuit with few gates. Here, we use Bshouty's learning algorithm, based on his *monotone theory* [8] as a basis for learning a CDNF representation of the target function, which leads to three levels in the computed circuits. This is still a low number, and thus allows to drive the resulting circuit with high clock rates, but offers a better chance for minimizing the number of gates. We start with an explanation of the necessary theory, and then describe how it can be applied when f and c are given in BDD form.

Let V be some set of variables and X, Y, and Z be subsets of V. We write $X \subseteq_Z Y$ if and only if $(X \cap (V \setminus Z)) \subseteq$

$(Y \cap (V \setminus Z))$ and $(X \cap Z) \supseteq (Y \cap Z)$. A Boolean function $f : 2^V \to \mathbb{B}$ is Z-monotone if for all $X, Y \subseteq V$, if $X \subseteq_Z Y$ and $f(X) = \mathbf{true}$, then $f(Y) = \mathbf{true}$.

Bshouty's learning algorithm represents the function-to-learn as a conjunction of Boolean formulas in disjunctive normal form. Each of this DNF formulas is Z-monotone for some $Z \subseteq V$. From the computational learning theory perspective [1], Bshouty's CDNF learning algorithm employs two types of *queries*: membership queries and equivalence queries. In this paper, we extend its idea by modifying the algorithm in order to make use of don't care variable valuations. Performing a membership query in this context then means that for a variable valuation X, we check if $f(X) \vee \neg c(X) = \mathbf{true}$. Performing an equivalence query means checking if the variable valuations that c maps to $\mathbf{true}$ are models of the candidate CDNF formula if and only if they are mapped to $\mathbf{true}$ by f. The following code describes the learning process:

```
1:  procedure LEARNCDNF($V, f, c$)
2:    $P = \emptyset$
3:    while true do
4:      $g := \bigwedge_{(h,d) \in P} h$
5:      if $(g \wedge (\neg f) \wedge c) \neq \mathbf{false}$ then
6:        $b :=$ pick some variable valuation in $g \wedge (\neg f) \wedge c$
7:        $P := P \cup \{(\mathbf{false}, b)\}$
8:      else if $((\neg g) \wedge f \wedge c) \neq \mathbf{false}$ then
9:        $b :=$ pick some variable valuation in $(\neg g) \wedge f \wedge c$
10:       for $\{(h, d) \in P \mid b \not\models h\}$ do
11:         $b' := b$
12:         for $v \in \{v' \in V \mid b'(v') \neq d(v')\}$ do
13:           $b'' := b'$
14:           $b''(v) := \neg b''(v)$
15:           if $f(b'') \vee \neg c(b'')$ then
16:             $b' := b''$
17:           end if
18:         end for
19:         $h' := h \vee \bigwedge_{v \in V, b'(v) \neq d(v)} \begin{cases} v & \text{if } b'(v) = \mathbf{true} \\ \neg v & \text{else} \end{cases}$
20:         $P := P \setminus \{(h, d)\} \cup \{(h', d)\}$
21:       end for
22:     else
23:       return $g$
24:     end if
25:   end while
26: end procedure
```

The algorithm maintains a candidate CDNF formula in P. Every DNF formula is stored together with its monotonicity base. Line 5 checks for *false-positives*, and line 8 for *false-negative* variable valuations. False-positives are valuations $X \subseteq V$ with $c(X) = \mathbf{true}$ that are models of the candidate formula, but for which $f(X) = \mathbf{false}$. Likewise, false-negatives are valuations X for which $c(X) = f(X) = \mathbf{true}$, but X is not a model of the candidate formula. Both witness the misclassification of a variable valuation. Whenever we find a false-positive X, we add a DNF that is kept X-monotone during the run of the algorithm. For a false-negative X, we update all DNFs with a cube (a conjunction of literals) that ensures that X becomes a model of the DNF. For this, we first make the false-negative as similar to the monotonicity base as possible without changing the fact that the circuit is allowed to output $\mathbf{true}$ for this valuation. Then we add a cube that contains only literals that point away from the monotonicity base. This way, the DNF formula remains d-monotonous with respect to its base d, but stays small at the same time. For more details on Bshouty's CDNF learning algorithm, the reader is referred to [28].

The algorithm terminates because in every iteration, either a false-positive or a false-negative is resolved, and the maximum number of potential misclassifications is finite. Note that resolving a false-positive will typically add new false-negatives because the newly added DNF is initially empty, i.e. **false**. However, resolving a false-positive X eliminates it once and for all. The reason is that the new DNF that is added is kept X-monotone. It is extended with cubes containing literals that point away from the monotonicity base only. Hence, X can never become a model of that DNF and $g(X)$ will always remain **false**. The algorithm is correct because upon termination, there are no more misclassifications.

In a post-processing step, we simplify the formula produced by LEARNCDNF. We remove all DNFs, cubes and literals for which removing leaves the CDNF consistent with f on c.

Example 2. *We apply* LEARNCDNF *to the learning problem* $(\{v_1, v_2\}, f, c)$ *from Example 1, defined by the following table.*

v_1	v_2	f	c	g^1	g^2	g^3
false	**false**	**false**	**false**	**true**	**false**	**false**
false	**true**	**false**	**false**	**true**	**false**	**true**
true	**false**	**false**	**true**	**true**	**false**	**false**
true	**true**	**true**	**true**	**true**	**false**	**true**

We have that $P^1 = \emptyset$, *so* $g^1 = \mathbf{true}$. *Since* $g^1 \wedge (\neg f) \wedge c$ *is satisfiable, there exists a false-positive* b^1 *with* $b^1(v_1) = \mathbf{true}$ *and* $b^1(v_2) = \mathbf{false}$. *To resolve it, a new DNF formula, initialized to* **false**, *is added to* P *together with its monotonicity base* b^1. *In the next iteration,* g^2 *is* **false**. *Consequently,* $g^2 \wedge (\neg f) \wedge c$ *is unsatisfiable, so there exists no false-positive. However, there is a false-negative* b^2, *defined as* $b^2(v_1) = b^2(v_2) = \mathbf{true}$. *To resolve it, the DNF formula* $h^2 = \mathbf{false}$ *is weakened with an additional cube. To get a small cube,* b^2 *is modified to match the monotonicity base* b^1 *as well as possible.* b^1 *and* b^2 *differ only in* v_2, *so this value is flipped to obtain* b''^2 *as* $b''^2(v_1) = \mathbf{true}$, $b''^2(v_2) = \mathbf{false}$. *However, for* b''^2 *it is not allowed to output* $\mathbf{true}$, *because* $f(b''^2) \vee \neg c(b''^2)$ *is* **false**, *so the flip is retracted. Since* b^1 *and* b^2 *differ only in* v_2 *and* $b'^2(v_2) = \mathbf{true}$, *the empty DNF in* h^2 *is extended to* v_2. *Since* P^3 *now contains only the DNF* v_2, g^3 *is* v_2 *in the next iteration. Using* g^3, *there is neither a false-positive nor a false-negative, so* $g^3 = v_2$ *is reported as solution.*

V. Experimental Results

In this section, we first briefly describe our implementation and experimental setup. Then we present our experimental results with the synthesis tools RATSY and UNBEAST.

A. *Implementation and Experimental Setup*

We implemented the learning algorithms in a circuit extraction tool that can be run in 9 different modes. For every mode, the following table summarizes the learning method, whether the basic method is complemented, whether variables are minimized using MINVARS, and the output format. We say that a method is complemented if we negate the function to learn before applying the learning algorithm. By duality, we can then use the CNF learning algorithm for obtaining a DNF result, and the CDNF learner to get a DCNF (disjunction of DNFs) output. Mode 8 is special: for every output, it applies the learning methods of modes 1, 3, 5, and 7, and picks the smallest implementation.

Mode	Learning Method	Compl.	MINVARS	Outcome
0	Bshouty	no	no	CDNF
1	Bshouty	no	yes	CDNF
2	Bshouty	yes	no	DCNF
3	Bshouty	yes	yes	DCNF
4	CNF Refinement	no	no	CNF
5	CNF Refinement	no	yes	CNF
6	CNF Refinement	yes	no	DNF
7	CNF Refinement	yes	yes	DNF
8	both	both	yes	all

As input, our tool takes a file containing a general strategy $\mathcal{S}$. As output, it produces a circuit in SMV or BLIF format. All symbolic computations are done using BDDs. We use CUDD [31] as BDD library with dynamic variable reordering enabled. Our tool is written in C++. The implementation as well as the input files and all scripts to reproduce the experimental results are available for download[3].

In our experiments, we run RATSY and UNBEAST to synthesize circuits for several specifications. We also export the general strategies and synthesize circuits with our learning-based extractor. ABC[4] 70930 is used to map circuits to standard cells. The gates in our standard cell library have a fan-in of at most 4 (which can increase the depth of the circuit). All circuits produced by our extractor have been successfully model-checked against their original specifications, using NUSMV 2.5.4 [12] with bounded model checking.

All experiments were performed on an Intel Xeon E5430 CPU with 4 cores running at 2.66 GHz, 64 GB of RAM, and a 64 bit Linux. All programs run single-threaded, so only one core was actually used. The maximum memory consumption of our new circuit extractor was 3.3 GB in our experiments.

B. *Experiments with* RATSY

RATSY's built-in circuit extractor uses the cofactor approach sketched in Section II. Table I compares this technique with our new approach. Due to space constraints, the comparison includes mode 1 and 7 (see Section V-A) only. These modes were chosen because they achieved good results. Results for the other modes can be found in Table III in the appendix. We evaluate the methods on three parametrized specifications. The first one defines an arbiter for ARM's AMBA AHB bus [6]. It is parametrized with the number of masters it can handle. These specifications are denoted as Ai, where i is the number of masters. The second specification, denoted A' i, is a less optimized variant of the former. It is described in [5]. The third specification is denoted by Gi and defines a generalized buffer [6] connecting i senders to two receivers. The bit numbers of the general strategies range from $|U| = 24$ and $|W| = 12$ (for G2) to $|U| = 129$ and $|W| = 63$ (for A15). Table III contains the exact numbers for every benchmark.

Column 1 in Table I lists the time needed by RATSY to turn the general strategy into a circuit. The size of the resulting circuit in terms of the total number of standard cells (gates plus flip-flops, but the flip-flops are typically negligible) is given in column 2. Column 3 contains the corresponding depth of the combinational circuit. The columns 4 to 7 show the results for circuit extraction with our extractor in mode 1. Column 4 gives the circuit extraction time. The columns 5 and 6 list the size and depth of the resulting circuits. Column 7 contains the circuit size improvement factor due to our method. The columns 8 to 11 show the same information for mode 7. Computation time entries preceded by a ">" indicate time-outs. A "-" stands for missing data due to a time-out. The suffix k stands for a multiplication of the respective number by $1\,000$. The table does not contain entries for A' i with $i > 5$ because for these specifications, RATSY did not finish within $100\,000$ seconds. The sums and averages in the last two lines only take into account benchmarks for which all methods terminate.

C. *Experiments with* UNBEAST

UNBEAST is a bounded synthesis tool that applies the Kukula/Shiple method (see Section II) for circuit extraction. The comparison with our new approach is summarized in Table II for the modes 1 and 7 of our new circuit extraction tool. Results for the other modes can be found in the appendix. For the comparison, we use the (realizable) LILY benchmarks [20], which are denoted as Li. For some of these benchmarks we created several variants. They are named Li–j, where j is a size parameter. We also use a specification for a load balancer [16] (the final version), which is parameterized by the number i of clients. These specifications are referred to as Bi. Table II is organized just like Table I. A circuit depth of 0 means that the combinatorial circuit could be implemented without gates, i.e., all outputs are either equal to an input or to the constants **true** or **false**. The bit numbers of the general strategies range from $|U| = 9$ and $|W| = 9$ (on L13) to $|U| = 93$ and $|W| = 92$ (on B5). The individual bit numbers can be found in Table IV in the appendix.

D. *Discussion*

Fig. 2 shows a scatter plot comparing the size of the circuits produced by our new extractor in mode 1 against those produced by RATSY or UNBEAST. On most benchmarks run through RATSY, an improvement of around one order of

[3]http://www.iaik.tugraz.at/content/research/design_verification/others/
[4]http://www.eecs.berkeley.edu/~alanmi/abc/

TABLE I
COMPARISON WITH RATSY.

Col.	1	2	3	4	5	6	7	8	9	10	11
	RATSY			Mode 1				Mode 7			
	time: code generation	circuit size	circuit depth	time	circuit size	circuit depth	shrinking factor	time	circuit size	circuit depth	shrinking factor
	[sec]	[#cells]	[-]	[sec]	[#cells]	[-]	[-]	[sec]	[#cells]	[-]	[-]
A2	1.2	733	18	1.4	269	5	2.7	1.3	260	5	2.8
A3	19	5.6 k	25	19	355	5	16	21	565	5	9.9
A4	67	10 k	33	461	1.8 k	7	5.5	480	3.7 k	8	2.7
A5	135	5.7 k	25	221	663	5	8.5	239	1.2 k	6	4.9
A6	204	8.2 k	26	233	819	6	10	251	1.3 k	6	6.4
A7	840	16 k	36	452	1.0 k	6	16	488	1.6 k	6	10
A8	6.6 k	135 k	46	>100 k	–	–	–	>100 k	–	–	–
A9	1.8 k	22 k	41	4.2 k	1.3 k	6	16	6.7 k	2.4 k	6	9.3
A10	3.0 k	19 k	44	8.9 k	1.5 k	6	12	7.0 k	2.4 k	7	7.6
A11	4.0 k	39 k	46	7.4 k	1.8 k	7	21	7.5 k	3.1 k	6	13
A12	10 k	38 k	50	16 k	2.0 k	6	19	37 k	3.1 k	7	12
A13	15 k	65 k	54	45 k	2.4 k	6	28	31 k	3.5 k	7	19
A14	15 k	47 k	42	36 k	2.6 k	7	18	83 k	3.6 k	7	13
A15	19 k	70 k	59	75 k	3.0 k	7	24	99 k	4.0 k	7	18
A′2	1.7	1.0 k	16	1.9	224	5	4.6	2.4	306	5	3.4
A′3	169	17 k	26	77	465	5	38	103	781	6	22
A′4	914	28 k	32	984	677	5	41	5.1 k	5.2 k	8	5.3
A′5	9.7 k	101 k	37	18 k	893	5	113	16	1.9 k	6	54
G2	0.1	249	11	0.1	53	3	4.7	0.1	65	3	3.8
G3	0.2	394	12	0.3	123	4	3.2	0.3	174	4	2.3
G4	0.5	721	18	0.5	119	3	6.1	0.5	262	5	2.8
G5	1.2	1.8 k	18	2.3	444	6	4.2	1.9	674	6	2.7
G6	7.7	6.2 k	22	6.8	1.1 k	6	5.8	2.1	828	6	7.5
G7	3.3	3.5 k	23	11	1.7 k	7	2.0	15	3.4 k	7	1.0
G8	8.1	5.8 k	26	2.6	278	4	21	7.4	5.5 k	8	1.1
G9	5.9	3.4 k	25	430	5.8 k	9	0.6	74	10 k	8	0.3
G10	14	6.5 k	29	8.0 k	22 k	9	0.3	57	13 k	9	0.5
G11	18	9.8 k	33	71 k	47 k	10	0.2	157	28 k	9	0.4
G12	35	14 k	34	>100 k	–	–	–	711	62 k	10	0.2
sum	80 k	531 k	827	292 k	100 k	160	5.3	294 k	101 k	173	5.3
avg.	3.0 k	20 k	31	11 k	3.7 k	5.9	16	11 k	3.7 k	6.4	8.7

TABLE II
COMPARISON WITH UNBEAST.

Col.	1	2	3	4	5	6	7	8	9	10	11
	UNBEAST			Mode 1				Mode 7			
	time: code generation	circuit size	circuit depth	time	circuit size	circuit depth	shrinking factor	time	circuit size	circuit depth	shrinking factor
	[sec]	[#cells]	[-]	[sec]	[#cells]	[-]	[-]	[sec]	[#cells]	[-]	[-]
L3	0.1	845	24	0.1	19	1	44	0.1	19	1	44
L3-6	0.3	6.3 k	86	15	2.6 k	6	2.4	0.9	79	3	80
L5	0.1	908	23	0.1	24	1	38	0.1	27	2	34
L6	0.1	2.5 k	40	0.1	38	3	65	0.1	42	2	59
L7	0.1	551	21	0.1	22	1	25	0.1	25	2	22
L8	0.1	113	11	0.1	10	0	11	0.1	10	0	11
L9	0.1	450	15	0.1	16	1	28	0.1	16	1	28
L10	0.1	1.4 k	35	0.1	22	0	64	0.1	22	0	64
L12	0.1	524	21	0.1	15	0	35	0.1	15	0	35
L13	0.1	28	7	0.1	9	1	3.1	0.1	9	1	3.1
L14	0.1	3.4 k	31	0.1	17	1	203	0.1	17	1	203
L15	0.1	277	15	0.1	16	2	17	0.1	16	2	17
L16	0.1	830	21	0.1	29	3	29	0.1	37	3	22
L17	0.1	1.5 k	37	0.1	21	1	71	0.1	25	2	60
L18	0.3	23 k	73	0.2	79	4	286	0.2	73	4	309
L19	0.1	531	26	0.1	21	1	25	0.1	21	1	25
L20	0.1	4.7 k	52	0.5	120	4	39	0.3	84	4	56
L21	0.4	13 k	70	2.6	114	5	112	3.6	205	5	62
L22	0.1	1.1 k	43	0.1	107	4	11	0.1	43	3	26
L22-5	0.1	950	35	0.1	104	4	9.1	0.1	40	3	24
L22-6	0.1	995	40	0.1	107	4	9.3	0.1	43	3	23
L22-7	0.1	1.2 k	43	0.1	107	4	11	0.1	42	3	28
L22-8	0.1	1.0 k	35	0.1	106	4	9.5	0.1	41	3	24
L22-9	0.1	910	35	0.1	105	4	8.7	0.1	41	3	22
L23	0.1	354	18	0.1	16	1	22	0.1	16	1	22
B2	0.1	3.5 k	51	0.1	35	2	99	0.1	43	2	81
B3	0.7	27 k	79	1.9	437	6	62	0.8	131	5	208
B4	5.5	171 k	151	1.1 k	6.5 k	9	26	3.5 k	23 k	9	7.4
B5	650	816 k	189	99 k	17 k	9	49	>100 k	–	–	–
sum	7.5	268 k	1.1 k	1.1 k	11 k	77	25	3.5 k	24 k	69	11
avg.	0.3	9.6 k	41	39	387	2.8	49	125	872	2.5	57

magnitude can be observed, with a tendency to greater improvements for larger circuits. For UNBEAST the improvement reaches almost two orders of magnitude on many benchmarks. Mode 8 (only included in the appendix) produces even smaller circuits, but at the costs of higher running times. In contrast to the many methods we have already tried (cf. Section II), these results are very promising. Table I and II also show a significant improvement of the circuit depths. For RATSY, the average is reduced from 31 to 6 after mapping to standard cells. For UNBEAST there is an average reduction from 41 to less than 3 in our experiments.

The downside of our new method is that computation times grow. For the RATSY benchmarks Ai and A′i, the slow-down factor is mostly below 4. Only for the benchmarks Gi, the circuit extraction times grow much faster with increasing i than with the cofactor approach implemented in RATSY. Also compared to the Kukula/Shiple method of UNBEAST, a considerable slow-down can be observed on the Bi benchmarks. On the Gi benchmarks, mode 7 appears to scale better than mode 1, but at the costs of producing larger circuits.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new approach for extracting circuits from general strategies that improves the circuit size by roughly one to two orders of magnitude, compared to previous techniques. Moreover, it reduces the depths of the resulting circuits, allowing them to be run at higher clock rates. General strategies are typical intermediate results of reactive synthesis workflows, and thus our contribution significantly increases the quality of the circuits computed in reactive synthesis.

During our quest for effective and efficient circuit extraction techniques that go beyond the cofactor approach of Bloem at al. [6], we tried a large number of older techniques from literature. Our experience shows that exploiting the large degree of non-determinism that we have in general strategies is not a simple task and techniques not geared towards such cases do not perform well. Our approach on the other hand is built around the idea of computational learning of Boolean

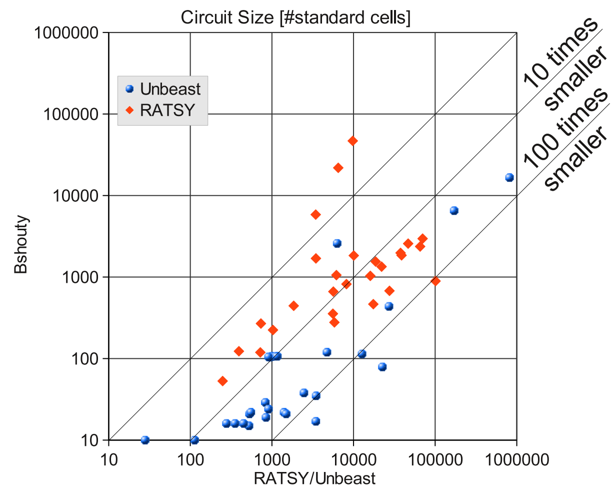


Fig. 2. Circuit size improvement.

functions. It allows exploiting non-determinism in a natural way, which is the reason for the effectiveness of our approach.

In the future, we plan to implement more learning algorithms, refine them with heuristics for selecting better false-positives, false-negatives, and variable orderings, and to implement the algorithms also with SAT-solvers instead of BDDs. Furthermore, we want to compare the produced circuits with manual implementations to see how much potential for optimizations still exists, and to get new inspirations.

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[2] M. Lewis B. Becker, R. Ehlers and P. Marin. ALLQBF solving by computational learning. In *Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 370–384. Springer, 2012.

[3] D. Bañeres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve Boolean relations. In *Design Automation Conference (DAC'04)*, pages 416–421. ACM, 2004.

[4] 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 *Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[5] 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 *Design, Automation and Test in Europe (DATE'07)*, pages 1188–1193, 2007.

[6] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007.

[7] 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.

[8] N. H. Bshouty. Exact learning Boolean functions via the monotone theory. *Electronic Colloquium on Computational Complexity (ECCC)*, 2(8), 1995.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[10] K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. In *Asia and South Pacific Design Automation Conference (ASP-DAC'07)*, pages 944–949. IEEE, 2007.

[11] Y.-F. Chen and B.-Y. Wang. Learning Boolean functions incrementally. In *Computer Aided Verification (CAV'12)*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[12] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.

[13] W. Craig. Three uses of the Herbrand-Gentzen Theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[14] R. Ehlers. Short witnesses and accepting lassos in ω-automata. In *Language and Automata Theory and Applications (LATA'10)*, volume 6031 of *LNCS*, pages 261–272. Springer, 2010.

[15] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.

[16] R. Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012.

[17] E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

[18] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 2012. 10.1007/s10009-012-0228-z.

[19] J. R. Jiang, H. Lin, and W. Hung. Interpolating functions from large Boolean relations. In *International Conference on Computer-Aided Design (ICCAD'09)*, pages 779–784. IEEE, 2009.

[20] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design (FMCAD'06)*, pages 117–124. IEEE, 2006.

[21] B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 258–262, 2007.

[22] R. Koenighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer Aided Design (FMCAD'11)*, pages 91–100. IEEE, 2011.

[23] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 113–123. Springer, 2000.

[24] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Synthesis and Simulation Meeting and International Interchange (SASIMI'92)*, pages 64–73, 1992.

[25] A. Mishchenko, S. Chatterjee, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.

[26] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Transactions on Computers*, 100(6):504–509, 1970.

[27] M. Schlaipfer, G. Hofferek, and R. Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In *Haifa Verification Conference (HVC'11)*, 2011. To appear.

[28] R. H. Sloan, B. Szörényi, and G. Turán. Learning Boolean functions with queries. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, 2010.

[29] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *Formal Methods in Computer-Aided Design (FMCAD'09)*, pages 77–84. IEEE, 2009.

[30] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 404–415. ACM, 2006.

[31] F. Somenzi. CUDD: CU decision diagram package, release 2.4.2, 2009.

APPENDIX

TABLE III
EXTENSIVE PERFORMANCE RESULTS USING RATSY. "T" INDICATES A TIME-OUT AFTER 100 000 SECONDS.

	RATSY				Mode 0		Mode 1		Mode 2		Mode 3		Mode 4		Mode 5		Mode 6		Mode 7		Mode 8	
	$\lvert U\rvert$	$\lvert W\rvert$	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
	[-]	[-]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]
A2	32	18	1.2	733	4.2	339	1.4	269	4.4	358	1.5	259	3.4	358	1.3	217	7.6	551	1.3	260	2.0	249
A3	41	23	19	5.6k	48	406	19	355	162	732	23	540	29	420	20	423	94	732	21	565	21	303
A4	48	26	67	10k	5.1k	2.5k	461	1.8k	65k	11k	1.8k	5.3k	4.4k	3.4k	123	1.2k	T	–	480	3.7k	2.7k	2.1k
A5	56	30	135	5.7k	921	703	221	663	2.5k	1.2k	235	824	717	850	233	756	1.4k	1.6k	239	1.2k	265	581
A6	63	33	204	8.2k	1.1k	835	233	819	5.7k	1.9k	260	1.3k	714	1.4k	292	960	3.3k	2.1k	251	1.3k	299	691
A7	71	37	840	16k	2.4k	1.1k	452	1.0k	8.7k	2.2k	498	1.6k	1.8k	1.8k	964	1.2k	9.8k	2.6k	488	1.6k	664	912
A8	78	40	6.6k	135k	T	–	T	–	T	–	T	–	T	–	T	–	T	–	T	–	T	–
A9	86	44	1.8k	22k	24k	1.3k	4.2k	1.3k	26k	3.4k	4.4k	2.1k	8.6k	2.7k	6.5k	1.6k	37k	3.4k	6.7k	2.4k	4.9k	1.2k
A10	93	47	3.0k	19k	T	–	8.9k	1.5k	T	–	7.0k	2.5k	24k	3.2k	9.3k	1.9k	T	–	7.0k	2.4k	13k	1.4k
A11	100	50	4.0k	39k	77k	1.9k	7.4k	1.8k	T	–	7.4k	2.9k	16k	3.9k	12k	2.2k	T	–	7.5k	3.1k	11k	1.6k
A12	107	53	10k	38k	T	–	16k	2.0k	T	–	20k	3.1k	T	–	48k	2.4k	T	–	37k	3.1k	36k	1.7k
A13	114	56	15k	65k	T	–	45k	2.4k	T	–	28k	3.6k	45k	5.3k	87k	2.7k	T	–	31k	3.5k	T	–
A14	121	59	15k	47k	T	–	36k	2.6k	T	–	38k	3.9k	58k	5.9k	T	–	T	–	83k	3.6k	92k	2.3k
A15	129	63	19k	70k	T	–	75k	3.0k	T	–	T	–	T	–	T	–	T	–	99k	4.0k	T	–
A'2	35	21	1.7	1.0k	4.7	298	1.9	224	5.2	400	2.2	265	3.6	365	1.6	188	8.4	540	2.4	306	2.4	183
A'3	43	25	169	17k	142	307	77	465	114	450	99	561	120	310	103	555	159	555	103	781	83	444
A'4	52	30	914	28k	T	–	984	677	T	–	4.1k	4.8k	61k	4.8k	1.7k	1.2k	T	–	5.1k	5.2k	1.1k	656
A'5	60	34	9.7k	101k	7.6k	466	18k	893	8.6k	842	12k	1.1k	6.7k	475	15k	900	11k	1.4k	16k	1.9k	9.1k	834
G2	24	12	0.1	249	0.3	79	0.1	53	0.4	81	0.1	61	0.2	80	0.1	57	0.2	87	0.1	65	0.1	56
G3	28	14	0.2	394	0.8	153	0.3	123	0.9	165	0.3	181	0.7	173	0.3	147	0.7	214	0.3	174	0.4	136
G4	32	16	0.5	721	1.7	140	0.5	119	2.0	169	0.5	190	1.7	149	0.5	111	1.7	267	0.5	262	0.5	119
G5	36	18	1.2	1.8k	11	261	2.3	444	9.2	356	2.5	673	11	332	1.7	387	20	565	1.9	674	3.3	268
G6	39	19	7.7	6.2k	20	502	6.8	1.1k	15	371	2.4	601	8.7	463	4.3	1.0k	14	617	2.1	828	5.8	212
G7	42	20	3.3	3.5k	41	483	11	1.7k	121	518	27	3.5k	179	668	5.3	1.4k	927	1.2k	15	3.4k	47	1.4k
G8	46	22	8.1	5.8k	22	417	2.6	278	28	483	18	3.9k	17	414	2.7	261	178	626	7.4	5.5k	21	278
G9	50	24	5.9	3.4k	249	872	430	5.8k	551	790	180	9.9k	22k	1.1k	91	5.2k	34k	1.4k	74	10k	644	1.4k
G10	53	25	14	6.5k	871	1.9k	8.0k	22k	418	790	95	9.8k	14k	1.3k	1.3k	19k	8.4k	1.3k	57	13k	3.6k	410
G11	56	26	18	9.8k	7.4k	3.4k	71k	47k	300	969	263	20k	1.4k	1.5k	3.7k	35k	T	–	157	28k	25k	470
G12	59	27	35	14k	740	2.2k	T	–	1.4k	1.2k	1.1k	44k	37k	1.7k	14k	75k	95k	1.6k	711	62k	T	–
avg.	62	31	3.0k	23k	5.8k	930	11k	3.7k	5.7k	1.3k	4.6k	4.7k	12k	1.7k	7.7k	6.0k	11k	1.1k	11k	5.8k	8.0k	794

TABLE IV
EXTENSIVE PERFORMANCE RESULTS USING UNBEAST. "T" INDICATES A TIME-OUT AFTER 100 000 SECONDS.

	UNBEAST				Mode 0		Mode 1		Mode 2		Mode 3		Mode 4		Mode 5		Mode 6		Mode 7		Mode 8	
	$\lvert U\rvert$	$\lvert W\rvert$	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
	[-]	[-]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]	[sec]	[cells]
L3	21	19	0.1	845	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19	0.1	19
L3-6	59	57	0.3	6.3k	1.2	115	15	2.6k	34	3.7k	0.9	90	1.2	118	2.5	2.7k	13	3.1k	0.9	79	18	89
L5	26	24	0.1	908	0.1	24	0.1	24	0.1	27	0.1	27	0.1	24	0.1	24	0.1	27	0.1	27	0.1	24
L6	34	32	0.1	2.5k	0.1	40	0.1	38	0.1	49	0.1	41	0.1	41	0.1	43	0.1	54	0.1	42	0.1	37
L7	24	22	0.1	551	0.1	22	0.1	22	0.1	25	0.1	25	0.1	22	0.1	22	0.1	25	0.1	25	0.1	22
L8	10	10	0.1	113	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10	0.1	10
L9	16	16	0.1	450	0.1	20	0.1	16	0.1	18	0.1	16	0.1	20	0.1	16	0.1	18	0.1	16	0.1	16
L10	22	22	0.1	1.4k	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22	0.1	22
L12	15	15	0.1	524	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15	0.1	15
L13	9	9	0.1	28	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9	0.1	9
L14	17	17	0.1	3.4k	0.1	17	0.1	17	0.1	21	0.1	17	0.1	17	0.1	17	0.1	21	0.1	17	0.1	17
L15	14	14	0.1	277	0.1	15	0.1	16	0.1	21	0.1	16	0.1	15	0.1	16	0.1	21	0.1	16	0.1	16
L16	18	18	0.1	830	0.1	39	0.1	29	0.1	47	0.1	41	0.1	40	0.1	28	0.1	48	0.1	37	0.1	29
L17	20	21	0.1	1.5k	0.1	33	0.1	21	0.1	35	0.1	25	0.1	36	0.1	21	0.1	37	0.1	25	0.1	21
L18	34	35	0.3	23k	1.3	249	0.2	79	3.0	465	0.2	85	1.0	224	0.2	70	1.9	451	0.2	73	0.3	83
L19	21	21	0.1	531	0.1	32	0.1	21	0.1	33	0.1	21	0.1	31	0.1	21	0.1	28	0.1	21	0.1	21
L20	28	29	0.1	4.7k	0.4	76	0.5	120	0.4	126	0.3	102	0.3	77	0.3	99	0.4	92	0.3	84	0.6	86
L21	32	32	0.4	13k	2.2	123	2.6	114	16	468	12	419	3.3	248	4.1	261	4.3	218	3.6	205	4.6	101
L22	28	26	0.1	1.1k	0.1	61	0.1	107	0.4	215	0.1	40	0.1	63	0.1	109	0.2	169	0.1	43	0.2	38
L22-5	24	22	0.1	950	0.1	53	0.1	104	0.3	205	0.1	38	0.1	59	0.1	105	0.1	175	0.1	40	0.1	34
L22-6	28	26	0.1	995	0.1	61	0.1	107	0.4	215	0.1	40	0.1	63	0.1	109	0.1	169	0.1	43	0.2	38
L22-7	27	25	0.1	1.2k	0.1	56	0.1	107	0.4	209	0.1	39	0.1	62	0.1	108	0.2	178	0.1	42	0.1	37
L22-8	26	24	0.1	1.0k	0.1	55	0.1	106	0.4	207	0.1	38	0.1	61	0.1	107	0.1	177	0.1	41	0.1	36
L22-9	25	23	0.1	910	0.1	54	0.1	105	0.4	206	0.1	39	0.1	60	0.1	105	0.1	176	0.1	41	0.1	35
L23	16	16	0.1	354	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16	0.1	16
B2	33	32	0.1	3.5k	0.2	45	0.1	35	0.5	113	0.1	44	0.2	54	0.1	33	0.2	66	0.1	43	0.2	36
B3	44	43	0.7	27k	4.2	453	1.9	437	7.9	280	1.6	165	2.5	467	1.1	447	1.1	169	0.8	131	3.3	160
B4	77	76	5.5	171k	T	–	1.1k	6.5k	T	–	T	–	19k	7.9k	264	8.1k	8.1k	22k	3.5k	23k	T	–
B5	93	92	650	816k	T	–	99k	17k	T	–	T	–	T	–	T	–	T	–	T	–	T	–
avg.	27	26	23	37k	0.4	64	3.4k	947	2.4	250	0.6	54	663	349	9.7	451	291	993	125	872	1.0	40

Automated Debugging of Missing Input Constraints in a Formal Verification Environment

Brian Keng
Dept. of Elec. and Comp. Eng.
University of Toronto
Toronto, Canada

Andreas Veneris
Dept. of Comp. Sci & Elec. and Comp. Eng.
University of Toronto
Toronto, Canada

***Abstract*—In the past decade, formal tools have increased functional verification efficiency by exhaustively searching for hard to find bugs. Often the counter-examples returned are not due to design bugs but due to missing constraints that are needed to model the surrounding environment. These types of false positives have become a great concern in the industry today. To address this issue, input constraints are typically added by the engineer to restrict the input space a formal tool is allowed to explore. These constraints are difficult to generate as they are usually implicit in the documentation or implementation of adjacent design blocks. As a consequence, this process reduces the efficiency of formal methodologies because missing input constraints must be determined before deep design bugs can actually be detected.**

In this work, we present an algorithm to automatically generate missing input constraints given a failing counter-example. The process begins by building a filtering function that models the failing behaviors from the counter-example. Next, using this function a list of fixed cycle properties are generated and filtered to return a set of candidate input constraints for use in debugging. Preliminary experimental results show that the generated properties provide a strong intuition as to what input constraints may be missing.

I. Introduction

Functional verification is one of the most time consuming steps in the VLSI design flow taking up to 46% of the total design time [1]. To ease this growing burden, new tools and technologies have been developed such as assertion-based verification (ABV). ABV has shown to improve observability and increase overall verification efficiency. Along with traditional simulation-based techniques, modern ABV flows make wide use of formal technologies.

Formal methods allow a user to exhaustively explore the state space of a design in an attempt to find corner case counter-examples that elude traditional simulation-based verification. In formal property checking, a design block is verified against a precisely defined formal property written in an assertion language such as SystemVerilog Assertions (SVA) or Property Specification Language (PSL). As such, when a formal verifier returns a counter-example, the expectation is that a design bug has been detected. Although ideal, reports from the industry indicate that many failures are due to missing constraints from the surrounding environment and not because of design errors [2]. In the context of this work, we refer to such a situation as a *false positive*. These false positive are typically caused by missing constraints that are built into the environment but not explicitly documented. This results in formal tools reporting a failure when, in fact, the design may work as intended for the given environment.

To solve this issue, constraints in the form of formal properties are added by the engineer to restrict the space in which the formal tool can explore. The purpose of these constraints is to precisely model the restricted input space allowing the formal tool to find "real" design bugs. However, this presents a large debugging challenge to the engineer who is asked to play a guessing game as to which constraints need to be added. Adding to this overhead, often these constraints are implicitly specified in the documentation or implementation of adjacent design blocks. In many cases, the time-consuming manual process needed to identify these missing input constraints dominates the formal verification process leading to reduced efficiency.

This situation of generating constraints has also appeared in other contexts. During constrained random simulation, the work in [3] automatically generates constraint properties to bias the stimulus generator towards missing coverage holes. In compositional verification [4], a key step is generating assumption properties in order to verify the correctness of components separately. Previous work [5]–[7] aims to automatically generate an assumption on the interface between two components with the goal of proving the target property. Additionally, generating environmental constraints for software model checking [8], [9] and reactive system synthesis [10], [11] have also been studied. In these situations, the techniques effectively generate constraints to accomplish their respective goals. However, none of them addresses the wide-spread pain of debugging missing input constraints in a formal hardware verification flow.

In this work, we present an algorithm that takes the first steps towards automated debugging of missing input constraints in a formal Register Transfer Level (RTL) verification flow. This algorithm automatically generates fixed cycle input constraints in the form of SystemVerilog properties from a failing formal counter-example. The benefit of these generated constraints is twofold. First, the constraints are generated efficiently from the counter-example without the need to rerun the entire formal flow thus providing feedback quickly in the verification cycle. Further, the constraints are in the form of simple properties that can aid debugging by either being directly used for the actual missing constraint(s), or indirectly used to give intuition about the failure. The key insight is that the engineer cannot be taken out of the debugging loop entirely. Instead, the algorithm aims to efficiently return easy to understand feedback to speed up the debugging of missing constraints.

The algorithm begins by using the time-unrolled counter-example and extracting all minimal correction sets with respect to the inputs of the design. This information is used to build a filtering function that encodes the incorrect input combinations that led to the failure in the counter-example. Next, a dictionary of fixed cycle properties is used to generate a list of candidate input constraints based on relevant signals from the counter-example. Each property on the list is then used in conjunction with the filtering function to generate a small SAT instance to determine if the property is a candidate for a missing constraint. The result is a set of input constraints that each can restrict the bad input behavior seen in the counter-example. Preliminary experimental results confirm the efficiency in generating the new properties as well as their ability to provide effective guidance as to what input constraints may be missing.

The remaining sections of this paper proceed as follows. Section II and Section III present background material and the proposed approach, respectively. Section IV presents experimental results and Section V concludes this work.

II. Preliminaries

A. Minimal Correction Sets and Unsatisfiable Cores

Given an unsatisfiable (UNSAT) Boolean formula ϕ in conjunctive normal form (CNF), an *UNSAT core* is a subset of clauses that are unsatisfiable. A *Minimal Unsatisfiable Subset* (MUS) is an UNSAT core where every proper subset is satisfiable (SAT). A *Minimal Correction Set* (MCS) is a minimal subset of clauses of ϕ such that removing the subset will result in ϕ being satisfiable. There exists a duality relationship between MUSs and MCSs as it is possible to compute the set of one from the other [12]. Using this relationship, one can calculate all MUSs from all MCSs.

Given an UNSAT CNF formula ϕ, MCSs can be computed by introducing a fresh variable to each clause called a *relaxation variable*. If the variable is active, then the clause is effectively removed from the problem. Using this idea, cardinality constraints [13] can be used to find all minimal sets of relaxation variables that make ϕ SAT. For each solution, the set of active relaxation variables correspond to an MCS. This idea has been used extensively in modern Max-SAT solvers [14], [15] to compute MCSs.

With respect to debugging, a MUS intuitively represents one way in which a counter-example can excite an error, traverse its effects through the design components and cause a failure at the observation points. In this view, clauses correspond to the counter-example, components of the design and target property. Alternatively, an MCS represents a minimal set of clauses related to components that are potentially erroneous. In other words, removing the components related to the MCS clauses is a potential way to "correct" the design. UNSAT cores and MCSs have been widely used in various debugging applications such as [16].

III. Debugging Missing Input Constraints

A. Extracting Failing Behaviors from a Counter-Example

In this subsection, we develop a methodology to quickly determine whether a candidate input constraint will prevent a failure from occurring. A naive way to detect this is to simply re-run the formal tool with the added candidate constraint. This can be very computationally intensive especially if multiple input constraint candidates need to be tested. Instead, we will generate an approximate solution to this process by generating a function that intuitively represents the disallowed input behaviors from the unrolled counter-example. More precisely, this function will represent all MUSs with respect to the input unit clauses of the unrolled counter-example. Using this function, potential input constraints can be efficiently checked to ensure that they do not cause a failure in a similar manner to the given counter-example.

Consider the CNF formula ϕ of the time-frame expanded circuit and the corresponding counter-example:

$$\phi = S \cdot X \cdot T \cdot P \tag{1}$$

where S represents the initial state, X the counter-example input vector, T the unrolled circuit transition relation, and P the property to be checked. Since ϕ models the counter-example of the unrolled circuit, it is guaranteed to be UNSAT.

Instead of computing all MUSs for ϕ to generate our desired function, a less expensive computation can be performed by examining only the inputs clauses from X. The intuition here is that we are only concerned with missing input constraints, so it is unnecessary to perform extra computation for finding all MUSs not relating to inputs.

More precisely, we wish to extract all minimal* subsets of input unit clauses from X (denoted by U^k for the k^{th} such set) such that $S \cdot T \cdot P \cdot U^k$ is UNSAT. This will allow us to build a function, F, that represents the disjunction of all MUSs with respect to the inputs, shown in the next equation:

$$F = U^0 + ... + U^k \tag{2}$$

Given a candidate input constraint, A, if $F \cdot A$ is SAT, then A does not prevent the failure given in the counter-example since at least one of U^k is SAT. Inversely, if $F \cdot A$ is UNSAT, then A will ensure that future failures will not occur in the same way as the given counter-example. However in the latter case, A may not constrain the input space enough to prevent all failures, but it at least prevents failures similar to those seen in the counter-example.

For the i^{th} literal in U^k, denoted by u_i^k, Equation 2 can be expanded to give:

$$\begin{aligned} F =& u_0^0 u_1^0 ... u_{|U^0|}^0 + ... + u_0^k u_1^k ... u_{|U^k|}^k \\ =& \overline{(\overline{u}_0^0 + \overline{u}_1^0 + ... + \overline{u}_{|U^0|}^0) ... (\overline{u}_0^k + \overline{u}_1^k + ... + \overline{u}_{|U^k|}^k)} \end{aligned} \tag{3}$$

* Minimal in the sense that removing any clause from U^k will make $S \cdot T \cdot P \cdot U^k$ become SAT.

Notice that when F evaluates to false, at least one literal in each U^k term is false. In other words, all U^k MUSs can be broken by negating at least one literal from each term in F. Correspondingly, ϕ can be made SAT if at least one literal from each term in F is negated for the respective unit clauses in ϕ. Further, removing a minimal set of the corresponding unit clauses from the original problem will give an equivalent effect. Define this minimal set to be $V^k \subseteq X$ for the k^{th} such set.

The set V^k can be thought of as the k^{th} MCS with respect to the input literals. In fact, the relationship between the minimal subsets of inputs to make ϕ UNSAT (U^k), and the minimal subsets of inputs that need to be removed to make ϕ SAT (V^k), is analogous to the relationship between MUSs and MCSs.

Using this relationship and the fact that these sets only contain unit clauses, F can be simplified further. Let the i^{th} literal in $V^k \subseteq X$ be denoted by v_i^k. Equation 3 can be simplified, by distributing the conjunctions and removing redundant terms/literals, to:

$$F = \overline{\overline{v}_0^0 \overline{v}_1^0 ... \overline{v}_{|V^0|}^0 + ... + \overline{v}_0^k \overline{v}_1^k ... \overline{v}_{|V^k|}^k} \quad (4)$$

Now each term of Equation 4 contains the conjunction of the negated literals of each V^k. Thus to build the function F, one only needs to find all V^k.

This can be accomplished in a similar manner to computing all MCSs. Begin by adding a fresh relaxation variable to each clause in X. Using cardinality constraints, find all minimal SAT solutions with respect to these relaxation variables similar to the process used by modern Max-SAT solvers [14], [15]. Each such solution will correspond to a V^k. After all such solutions are found, construct a SAT instance of the form $F \cdot A$, where A is the given input constraint to be checked. This instance checks whether A can restrict the input space to prevent a failure similar to the one seen in the counter-example.

Although computing MCSs can be computationally intensive in general, the proposed method only calculates them with respect to the input unit clauses. This allows the method to be more efficient as shown in the experimental results.

Example 1 *Consider the implementation of the simple state machine shown in Figure 1 that implements a modulo-2 counter that counts up when $a = 1$ and resets if $b = 1$. The property to be verified is:*

```
P: s == 2'b01 && a |=> s == 2'b10
```

Informally, if the counter is at `01` *and a is high, then in the next cycle it should be at* `10`*. If sent to a formal property checker, the property will fail because the property was written under the assumption that the reset signal b does not go high. A two cycle counter-example to this property is $X =< (a^0, \overline{b^0}), (a^1, b^1) >$, where the superscripts indicate the clock-cycle. Solving for all V^k, we find: $V^0 = \{a^0\}, V^1 = \{\overline{b^0}\}, V^2 = \{a^1\}, V^3 = \{b^1\}$. Which can directly be used to build $F = \overline{\overline{a^0} + b^0 + \overline{a^1} + \overline{b^1}}$.*

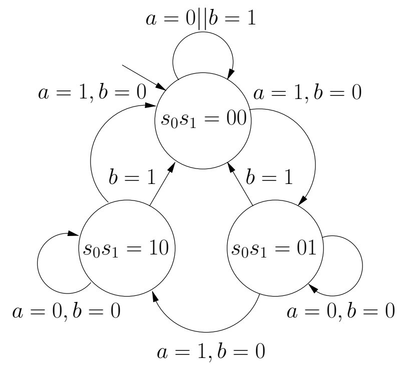


Fig. 1. Example 1: A Simple Modulo-2 Counter

B. Generating Fixed Cycle Properties

Missing constraints can be arbitrarily complex properties ranging from constant values to complex bus protocols that depend on the specifications. In general, there is no automated method to precisely generate these missing constraints that model the external environment. Even in cases where it may be possible, it is usually not practical. This is because algorithmically computed properties will likely be in some complex form that is unintelligible to user. This limits the benefit of any such technique to the user.

Instead, we take a different approach where simple fixed cycle properties are generated to give guidance to the user. In this way, the feedback can be used in conjunction with the user's knowledge to determine the missing input constraint, which frequently requires higher level design semantics. These properties may not be able to model all the complexities of the surrounding environment in all cases. However, the benefit of the proposed approach lies in the fact that it points the user to what types of constraints may be needed. Note that a more comprehensive set of properties can be used to expand upon the simple models presented in this preliminary study to gain greater benefit.

The process begins by selecting which input signals are involved in the counter-example failure. Any signal whose bit is used in F is considered to be a candidate for use in a generated property. Here, signals are categorized either as single-bit or multi-bit based upon the definition in the RTL.

For single bit signals and bits composing multi-bit signals, denoted by `a`, the following family of properties are generated:

- Stuck-at properties: `!a` and `a`.
- Hold: `$past(a) == a`, `$rose(a) |=> a`, `$rose(a) |=> !a`, `$fell(a) |=> a`, `$fell(a) |=> !a`

This family comprises of simple stuck-at properties and hold properties. These types of properties can be useful for detecting many different types of issues such as setting incorrect modes, or writing incorrect data.

Next, these multi-bit properties provide detection for common bus constraints such as one-hot, or incorrect addresses. `b1` and `b2` represent multi-bit signals, while `<val>` represents an assignment to the respective signal seen when simulating the counter-example. The following is a family of multi-bit properties:

- One-hot properties: `$onehot(b)` and `$onehot0(b)`.
- Equality operators: `b1 <op> <val>` and `b1 <op> b2`. Where `<op>` is one of {`<, <=, ==, >=, >`}, and where the size of `b1` and `b2` match.

These are slightly higher-level properties that may give intuition about certain missing constraints.

Once a list of properties are generated, each one can be efficiently filtered, as described in Section III-A, by creating a small SAT instance $F \cdot A$. Each instance is significantly smaller than the original unrolled circuit, allowing an efficient means of filtering these potential constraints without having to do an entire formal check.

Example 2 *Consider the filtering function F generated from Example 1 and the four stuck-at fault properties that would be generated: a, $\overline{a}$, b, and $\overline{b}$. Of these, only the first one would be filtered out since it would return SAT when run with F, while the others all return UNSAT. Of the remaining, it is easy to see how they translate to high-level behavior of the design: $\overline{a}$ prevents the counter from incrementing (a vacuous condition), b continually resets the machine (also vacuous), and $\overline{b}$ turns off reset (desired result).*

IV. Experimental Results

This section presents preliminary experimental results for the proposed approach. All experiments are run on a single core of a Intel Core i5 3.1 GHz quad-core workstation with 8 GB of RAM. Three designs are selected for our evaluation. The first two designs are from OpenCores [17] (`hpdmc`, `spi`), while the last one is a DDR2 controller from the OpenSparc project (`ddr`) [18]. For the OpenCores designs, SVA assertions are written based upon the accompanying design documentation. For the DDR2 controller, assertions from [19] are used which are based on the DDR2 specifications. These assertions are formally verified against the design using a commercial formal property checker [20], and any failures are considered instances of missing constraints. Each failing assertion is considered separately and is labeled by adding a number to the suffix of the circuit name.

Using these instances, our experimental methodology proceeds as follows. First, for each failing assertion, a counter-example is generated using a formal property checker. Next, the proposed approach from Section III uses the counter-example to generate a filtered list of missing constraints. `Minisat` [21] is used to solve all SAT instances, including generating the filtering function F. Finally, to check if any of the generated properties can be used as actual missing constraints, each property is re-run in a separate formal check with the original failing assertion. The comprehensive results for each instance are shown in Table I.

The first four columns of Table I show the instance name, number of gates, number of state elements, and counter-example length. The next three columns list the overall run-time in seconds of the proposed approach, which includes creating the filtering function as well as filtering, along with the original number of generated properties candidates from Section III-B, followed by the number remaining after filtering with function F. From the filtered list, the last three columns show the total run-time, number of non-vacuous passing instances and vacuous passing instances when re-running all generated constraints separately with the formal tool.

Overall, the results show that the filtering function can significantly reduce the number of candidates constraints from an average of 166 properties in column 6, down to an average of 24 in column 7 after filtering. Moreover, this is done with relatively little run-time making it ideal for fast analysis for use when debugging missing constraints. Compared to running each generated constraint in a separate formal check (column 8), the proposed method shows a 33.4x speedup on average. The last two columns show that in certain cases (e.g. `hpdmc` and `spi`), the simple properties can generate an exact constraint to prevent the failing assertion. Although in the case of `ddr`, none of the generated properties are able to prevent the failing assertion.

This is not a big surprise considering the simplicity of the generated constraints. However, a main point of this work is to aid debugging of missing constraints, not necessarily generate the exact constraint for the user. The simplicity of the generated constraints in this case is beneficial since it gives a intuitive method for the user as to which constraint is potentially missing. To further illustrate this point, we describe in detail the results of several cases from Table I.

Consider the first failing property for `hpdmc1` that specifies that after a read, an acknowledge signal should be asserted several cycles later based on the `tim_cas` register.

```
P: $rose(read) |-> (!tim_cas ##5 $rose(ack))
                   or (tim_cas ##6 $rose(ack))
```

The proposed approach generates 29 constraints, which deal primarily with bus and address line input pins. In particular, these generated constraints seemed relevant:

```
A1: wbc_adr_i[3:2] == 2'b00
A2: !wbc_we_i
A3: wbc_dat_i[6]
A4: !wbc_dat_i[6]
```

The first two constraints force `tim_cas` not to be over-written during programming of the control registers, while the last two[†] ensure that regardless of what is programmed, `tim_cas` should be held stable. These constraints give intuition that the `tim_cas` register should be held constant when checking this property.

For `spi1`, the assertion is a simple property to detect that the internal FIFO raises the `empty` flag correctly:

```
P: (re && (rp+2'h1)==wp) |=> empty;
```

The proposed approach generated 10 constraints, where the following were of particular interest:

[†] The reason that both `wbc_dat_i[6]` and its complement are suggested is that it ensures that the signal is held constant throughout the trace so that it does not toggle.

TABLE I
AUTOMATED GENERATION OF MISSING CONSTRAINTS EXPERIMENTAL RESULTS

instance info				**algorithm**			**check**		
instance name	# gates	# states	c-ex len	time (s)	cand	filter	time (s)	passing	vacuous
hpdmc1	9794	430	13	25	211	29	716	11	2
hpdmc2	9794	430	12	58	325	45	984	1	5
hpdmc3	9794	430	2	1	14	5	46	3	1
spi1	1724	132	4	1	40	10	80	1	8
spi2	1724	132	21	4	82	40	169	0	10
ddr1	55069	2474	9	248	310	20	3477	0	0
ddr2	55069	2474	6	42	180	20	1869	0	0

```
A1: adr_i[1:0] != 2'b10
A2: !we_i
```

These constraints attempt to disable writing to the FIFO. In this case, the assertion was written under the assumption that the writes cannot happen if the current operation has not been acknowledged yet, as given by this property: `!ack_o |=> !we_i`. In this case, the missing constraint involves a more complex protocol that is dependent on an output pin, `ack_o`. Despite this, the returned constraints can remind the user that this protocol should be followed.

In the case of `ddr1`, the assertion involves a more complex setup described in [19] to reach the target property of: "No more than 4 *activate* commands may be issued to the DDR2 SDRAM within a window of t_FAW clock cycles." All the returned constraints deal with the `other_que_pos` signal, which controls the `we` signal, which in turn causes an *activate* command. The work in [19] suggests that the issue is either a design error, or a constraint is missing to model an adjacent block that enforces this behavior. In the latter case, `other_que_pos` constraints point to this conclusion.

V. CONCLUSION

In this work, an algorithm is proposed that automatically generates missing input constraints from a failing counter-example. It begins by building a filtering function that models the failing behaviors from the counter-example. Next, a list of fixed cycle properties are generated and filtered to return a set of constraints that restrict the failing behavior in the counter-example. Preliminary experimental results show that the constraints can be efficiently generated and they provide effective guidance to improve the formal verification flow.

REFERENCES

[1] H. Foster, "Applied assertion-based verification: An industry perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009.

[2] A. Matsuda. (2011, May.) Overcoming the challenges of formal verification and debug. [Online]. Available: http://www.eetimes.com/design/eda-design/4216119/Overcoming-the-challenges-of-formal-verification-and-debug

[3] H.-H. Yeh and C.-Y. R. Huang, "Automatic constraint generation for guided random simulation," in *ASP Design Automation Conf.*, 2010, pp. 613–618.

[4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Psreanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2003, pp. 331–346.

[6] A. Gupta, K. L. Mcmillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design: An International Journal*, vol. 32, no. 3, pp. 285–301, June 2008.

[7] Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang, "Automated assume-guarantee reasoning through implicit learning," in *Computer Aided Verification*, 2010, pp. 511–526.

[8] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki, "DC2: A framework for scalable, scope-bounded software verification," in *Automated Software Engineering*, 2011, pp. 133 –142.

[9] S. Joshi, S. K. Lahiri, and A. Lal, "Underspecified harnesses and interleaved bugs," in *Principles of Programming Languages*, 2012, pp. 19–30.

[10] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Int'l Conf. on Formal Methods and Models for Codesign*, 2011.

[11] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment Assumptions for Synthesis," in *Int'l Conf. on Concurrency Theory*, 2008.

[12] M. H. Liffiton and K. A. Sakallah, "On Finding All Minimally Unsatisfiable Subformulas," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2005, pp. 173–186.

[13] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," in *JSAT*, vol. 2, 2006, pp. 1–26.

[14] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Test in Europe*, 2008, pp. 408–413.

[15] M. H. Liffiton and K. A. Sakallah, "Generalizing Core-Guided Max-SAT," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 481–494.

[16] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah, "Improved Design Debugging Using Maximum Satisfiability," in *Formal Methods in CAD*, 2007.

[17] OpenCores.org, 2007. [Online]. Available: http://www.opencores.org

[18] OpenSparc, 2012. [Online]. Available: http://www.opensparc.net

[19] A. Datta and V. Singhal, "Formal Verification of a Public-Domain DDR2 Controller Design," in *VLSI Design*, 2008, pp. 475–480.

[20] Cadence Design Systems, "Incisive Formal Verifier," 2012. [Online]. Available: http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx

[21] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

Algorithms for Software Model Checking: Predicate Abstraction vs. IMPACT

Dirk Beyer
University of Passau, Germany

Philipp Wendler
University of Passau, Germany

***Abstract*—CEGAR, SMT solving, and Craig interpolation are successful approaches for software model checking. We compare two of the most important algorithms that are based on these techniques: lazy predicate abstraction (as in BLAST) and lazy abstraction with interpolants (as in IMPACT). We unify the algorithms formally (by expressing both in the CPA framework) as well as in practice (by implementing them in the same tool). This allows us to flexibly experiment with new configurations and gain new insights, both about their most important differences and commonalities, as well as about their performance characteristics. We show that the essential contribution of the IMPACT algorithm is the reduction of the number of refinements, and compare this to another approach for reducing refinement effort: adjustable-block encoding (ABE).**

***Index Terms*—Formal Verification, Software Model Checking, Predicate Abstraction, Lazy Abstraction, Refinement Techniques, Interpolation, Large-Block Encoding**

I. Introduction

Software model checking has been successful for improving the quality of computer programs [4]. Several fundamental concepts were invented in the last decade which made it possible to scale the technology from tiny examples to real programs. Predicate abstraction [16] with counterexample-guided abstraction refinement (CEGAR) [14] and lazy abstraction [19] is one such technique. It was made popular by the tools SLAM [6] and BLAST [9], and is implemented in a number of other tools. Lazy abstraction with interpolants [20] is another approach, which is implemented in the tools IMPACT, WOLVERINE [24], and UFO [3]. More than half of the participants in the first competition on software verification [7], and almost all of those that are not based on bounded model checking, use one of these two concepts. Thus, we are interested in comparing the two concepts with each other, identifying their essential differences, and potentially learning new insights from them.

The contribution of our work is to systematically compare the two approaches. First, we re-implemented the IMPACT algorithm within the CPACHECKER framework. This is necessary in order to compare predicate abstraction with the IMPACT algorithm in the same framework: with the same parser frontend, SMT solver, and run-time environment. This verifies that our re-implementation shows all known characteristics in the comparison. Second, we present a unifying framework for predicate-based software model checking with an algorithm that can be configured (parametrized) such that it works like BLAST's predicate abstraction or IMPACT's approach. We show that the framework causes almost no overhead and the algorithms —when expressed in our framework— perform similarly to their original versions.

Now, we can conceptually and experimentally identify the differences of the algorithms. A performance comparison of our implementations of both algorithms (in the unified framework) shows that the key advantage of the IMPACT algorithm is the *forced covering* optimization that was presented by McMillan together with the algorithm [20]. This optimization effectively reduces the number of refinements and leads to a significant performance boost. However, without this optimization IMPACT does not perform better than predicate abstraction.

Another technique that has been shown to effectively reduce the number of refinements is adjustable-block encoding (ABE) [12] (a generalization of large-block encoding [8]), which was originally presented for predicate abstraction. We do not only compare the IMPACT algorithm to predicate abstraction with ABE, but also experiment with the combination of the IMPACT algorithm and ABE.

Availability of Data and Tools. We implemented all presented approaches (where not already existing) in the open-source verification framework CPACHECKER [11]. All experiments are based on publicly available benchmark programs from the last competition on software verification [7]. Our extensions of CPACHECKER are available under the Apache 2.0 license in the project repository via `http://cpachecker.sosy-lab.org`. Tables with our detailed results, as well as all benchmark programs, the configurations files, scripts, and a ready-to-run version of CPACHECKER are available on the supplementary webpage `http://www.sosy-lab.org/~dbeyer/cpa-uni`.

Related Work. A different approach to combine predicate abstraction and the IMPACT algorithm was presented by Albarghouthi et al. [1]. Their algorithm is similar to the IMPACT algorithm, but optionally computes an abstraction using predicates from previous refinements when creating new abstract states, instead of always setting these states to $true$. Furthermore, this approach represents the program counter symbolically (not explicitly), and does a single refinement for all error paths after the control-flow graph (CFA) has been completely unrolled into an abstract reachability graph (ARG) (instead of doing a separate refinement whenever a path to the error location was found). McMillan presented an application of the IMPACT principle to testing [21] and similarly proposed computing predicate abstractions in order to speed up the convergence of the algorithm.

Ermis et al. presented a technique for software verification which is also based on interpolation [15]. Instead of unrolling the CFA into an ARG by iteratively creating new abstract states at a frontier, they start with a path to the error location in the CFA and split all nodes along this path into two nodes, labeling one with the interpolant computed for this node, and the other with its negation [13]. Afterwards, all transitions between the newly created nodes and their neighbors are checked for feasibility and removed if appropriate. This is continued until no infeasible path to the error location is left. The authors compared their algorithm with the IMPACT algorithm, and applied large-block encoding to it. Complementing this work, we compare IMPACT with predicate abstraction. Ermis et al. support programs in the programming language Boogie and the tool cannot be directly applied to C benchmarks.

Extensions of the IMPACT approach have also been presented by Heizmann et al. [17] and Albarghouthi et al. [2]. These works add support for recursive programs.

II. BACKGROUND

We briefly provide some basic notions and concepts from the literature [9], and describe the two algorithms.

Programs. We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[1] A *program* is represented by a *control-flow automaton* (CFA), which consists of a set L of program locations (models the program counter l), an initial program location l_0, and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models the operation that is executed when control flows from one program location to another). The set of program variables that occur in operations from Ops is denoted by X. A *concrete state* of a program is a variable assignment $c : X \cup \{l\} \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete states of a program is denoted by C. A set $r \subseteq C$ of concrete states is called a *region*. Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \xrightarrow{g} c'$. A concrete state c_n is *reachable* from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \xrightarrow{g} c_i$.

Lazy Predicate Abstraction. Predicate abstraction in combination with CEGAR and lazy abstraction is a forward reachability analysis that unrolls the CFA into an abstract reachability graph (ARG) until a fixed point is reached. Abstract states are represented using predicates over program variables from a given set (the *precision*), which is initially empty. An abstract state is created by computing a boolean combination of these predicates that over-approximates the reachable concrete states. This abstraction computation is done using an SMT solver. When an abstract state that belongs to the error location is discovered, the concrete program path that leads to this state is reconstructed from the ARG and checked for feasibility. If the concrete path is infeasible, the current counterexample is said to be spurious, and the precision of the analysis needs to be refined in order to rule out this counterexample. This is done by computing a Craig interpolant [18] for each location on the path. The predicates contained in these interpolants are then added to the precision, and the analysis is restarted. This guarantees that all necessary predicates for proving program safety will be automatically discovered. For improved performance, the previously computed ARG is not completely deleted after refinement, but only those parts that need to be, are re-computed. Furthermore, the new predicates will not be used globally for all abstraction computations, but only in the part of the ARG and only at those locations of the CFA, for which they are relevant. The analysis terminates if either a non-spurious counterexample is found, or a fixed point is reached during unrolling the ARG (in which case the program is safe). In order to speed up the coverage checks between abstract states (which are necessary for determining whether the fixed point was reached), binary decision diagrams (BDDs) are used for representing the abstract states. This approach corresponds, e.g., to what is implemented in BLAST.

Lazy Predicate Abstraction with Adjustable Block-Encoding. Adjustable block-encoding (ABE) [12] aims at improving the performance of predicate abstraction by reducing the number of abstraction computations and refinements. It does not compute an abstraction for each new abstract state, but instead it groups abstract states into blocks and computes abstractions only once per block (at the end). Abstract states are now tuples of an abstract-state formula and a concrete path formula. The path formula of any abstract state always represents a set of concrete paths from the block entry to the location of this state. When a new state is created, the strongest post-condition of the previous state and the current edge is created and used as the concrete path formula. The abstract-state formula is copied from the previous state. If there exists already an abstract state with the same location inside the same block, both states are merged into one state by taking the disjunction of their path formulas. Only at the block end, an abstraction of the conjunction of the abstract-state formula and the concrete path formula of the current state is computed and used as the new abstract-state formula. The path formula is reset to *true* at the block end. ABE does not only reduce the number of abstraction computations, but also the number of coverage checks (which are only done at block ends), and the size of the ARG (due to merging of abstract states). The latter is the reason for a vastly reduced number of refinements. During refinement, interpolants are computed only for those abstract states at the block ends, because only for those states, predicates are needed for computing abstractions.

The block size can be freely chosen in ABE and does not need to be statically fixed (as in LBE [8]). If the block size is restricted to one single CFA edge (we name this *single-block encoding*, or SBE), an abstraction is computed

[1]Our implementation is based on CPACHECKER [11], which supports C programs in the CIL [22] subset of C and interprocedural program analysis.

for every new abstract state and the analysis behaves exactly like predicate abstraction in BLAST. Experiments have shown that for a good performance, the program structure should be taken into account when defining the block encodings. A good configuration is for example to define block ends at loop head locations of the program (ABE-Loops), such that the blocks will be the largest loop-free subgraphs of the CFA. Another suitable configuration with somewhat smaller blocks is to define block ends not only at loop heads but also at function entry and exit points (ABE-LF). This configuration is similar to large-block encoding [8]. ABE is implemented, for example, in CPACHECKER.

IMPACT *(Lazy Abstraction with Interpolants).* The IMPACT algorithm [20] similarly creates an unwinding of the CFA. However, it never performs abstraction computations, and instead initializes all new abstract-state formulas to $true$. This is similar to how predicate-abstraction algorithms work while the precision is still empty.

The algorithm consists of three basic steps, which are applied until no further change can be made. In theory, the steps can be executed in any order, but the right strategy is crucial for good performance. The steps are:

Expand(e). If the state e has no successors (i.e., it is a sink in the ARG) and is not covered, create the successor states using $true$ as their initial state formula, and add them to the ARG.

Refine(e). If e is an abstract state at the error location with a state formula different from $false$, compute inductive interpolants for the path from the ARG root to this state. For each state along this path, the state formula is strengthened by conjunctively adding the corresponding interpolant, and the state is marked as not covered. If the error path is infeasible, the state formula of the state at the error location is guaranteed to be $false$ (which marks unreachable states) after this step.

Cover(e_1, e_2). In this step, a state e_1 is marked as *covered* by another state e_2 if the following properties hold:

- e_2 is (and all of its ancestors are) not covered,
- both states belong to the same program location,
- the state formula of e_1 implies the one of e_2, and
- e_1 is not an ancestor of e_2.

If e_1 gets marked as covered, then (1) all states that are covered by e_1 or e_1's children are uncovered, and (2) all children of e_1 are implicitly considered as covered. Note that covered states never cover any other states themselves, i.e., no chains of coverage exist. In order to prevent an infinite loop of coverings and uncoverings, the step *Cover* may be applied only to pairs (e_1, e_2) where e_1 was created after e_2 (only older states can cover newer states, not vice versa).

The application order of the steps as proposed by McMillan is to expand nodes in a depth-first search. During the search, he keeps the invariant that the currently being expanded state and all its ancestors are not coverable by any other state (otherwise the current state would not need to be expanded). As soon as a state is found that belongs to the error location, the refinement procedure is run for this state. After a successful refinement, the invariant that no state on the path from the ARG root to the current state is coverable, is re-established by trying to cover all these states. (This can be optimized by checking only those states that have been strengthened during refinement.) This algorithm corresponds to the core algorithm of IMPACT, as presented by McMillan [20]. It is not available in IMPACT without the following optimization.

IMPACT *with Forced Covering.* When a new state is created in the IMPACT algorithm, its state formula is always $true$ and thus it can only be covered by another state at the same location with the same state formula. However, after some refinements, most states are expected to have stronger formulas, and thus coverage is unlikely, causing a large number of expansions and abstract states. As an optimization, one can try to strengthen the state formula of a new state such that this state can be covered by an existing state at the same location. This is called *forced covering*. In order to forcefully cover a state e_1 by another state e_2, the path from the nearest common ancestor of both states to e_1 is considered. If it can be proven that the state formula of e_2 holds at the location of e_1 after following this path from the nearest common ancestor, the state formula of e_2 can be set as the state formula of e_1. Thus, e_1 is immediately covered by e_2. Additionally, the states along the path from the nearest common ancestor to e_1 are strengthened by computing Craig interpolants for this path. This corresponds to the algorithm used for the benchmarks in the IMPACT article and to the tool implementation [20].

III. UNIFYING ALGORITHM

We formalize our unifying algorithm using the framework of *configurable program analysis* (CPA) [10]. A CPA specifies —independently of the analysis algorithm— the abstract domain and a set of operations that control the program analysis. Such a CPA can be plugged in as a component into the software-verification framework without the need to work on program parsers, exploration algorithms, and their general data structures. A CPA $\mathbb{C} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$ (which computes abstract successor states), a merge operator merge (which specifies if and how to merge abstract states when control flow meets), and a stop operator stop (which determines whether an abstract state is covered by another abstract state). The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semi-lattice $\mathcal{E}$ over abstract-domain elements (i.e., abstract states), and a concretization function that maps each abstract-domain element to the represented set of concrete states.

Using this framework, program analyses can be composed of several component CPAs. For example, we have defined and implemented separate CPAs for tracking the program counter, the call stack, and the successor-predecessor relationship of the ARG. Thus, we do not need to specify these aspects when defining a new core analysis.

Analysis Algorithm. We use the CPA algorithm for reachability analysis, which gets as input a CPA and two sets of abstract states: one is the set R_0 ($\mathsf{reached}$) of reachable abstract states, and one is the set W_0 ($\mathsf{waitlist}$) of abstract states that the algorithm is told to process next. The algorithm loops until the set waitlist is empty (all abstract states completely processed) and returns the two sets reached and waitlist. In each iteration, the algorithm takes one state e from the waitlist, computes all abstract successors and processes each of them. The algorithm checks if there is an existing abstract state in the set of reached states with which the new state is to be merged (e.g., at join points where control flow meets after completed branching). If this is the case, then the new, merged abstract state is substituted for the existing abstract state in both sets reached and waitlist. The stop operator ensures that a new abstract state is inserted into the work sets only if this is needed, i.e., the state is not already covered by a state in the set reached.

In order to be able to use CEGAR, we modify this existing CPA algorithm such that it terminates whenever a target state (a state at the error location) is encountered. First, we run the algorithm with singleton sets containing the initial state as input. If the algorithm terminates with a non-empty waitlist (target state found), we start the refinement procedure, which may modify both sets. Then, if the refinement was successful (i.e., the counterexample was infeasible), we run the CPA algorithm again with the modified sets as input. The analysis terminates if either the CPA algorithm finishes due to an empty waitlist ('safe'), or the refinement procedure determines that a feasible error path was found ('bug').

Another modification of the algorithm is necessary to support forced covering. We define a new operator fcover, which is called before an abstract state is going to be expanded. It takes as input the current state and the set of reached states, and returns a new set of reached states. This operator may change the set reached only by strengthening some states if this leads to the current state being covered afterwards. If the current state is still in the set reached afterwards (i.e., was not replaced by a strengthened version), then this state is explored, otherwise we continue with the next state from the waitlist.

The modified algorithm is shown as Algorithm 1. Our changes can be seen in lines 4–5 and 17–18. The shortness and simplicity of these modifications show that using the CPA framework as basis for new approaches is a good idea.

Algorithm 1 $\mathit{CPA}_{\mathsf{fcover}}(\mathbb{D}, R_0, W_0)$

```
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
       a forced covering strategy fcover,
       a set $R_0 \subseteq E$ of abstract states,
       a subset $W_0 \subseteq R_0$ of frontier abstract states,
       where $E$ denotes the set of elements of the semi-lattice of $D$
Output: a set of reachable states and a subset of frontier states
Variables: two sets reached and waitlist of elements of $E$
1:  reached := $R_0$; waitlist := $W_0$;
2:  while waitlist $\neq \emptyset$ do
3:    choose $e$ from waitlist; remove $e$ from waitlist;
4:    reached = fcover($e$, reached);
5:    if $e \in$ reached then
6:      for each $e'$ with $e \rightsquigarrow e'$ do
7:        for each $e'' \in$ reached do
8:          // Combine with existing abstract state.
9:          $e_{new}$ := merge($e', e''$);
10:         if $e_{new} \neq e''$ then
11:           waitlist := (waitlist $\cup \{e_{new}\}) \setminus \{e''\}$;
12:           reached := (reached $\cup \{e_{new}\}) \setminus \{e''\}$;
13:       // Add new abstract state?
14:       if $\neg$ stop($e'$, reached) then
15:         waitlist := waitlist $\cup \{e'\}$;
16:         reached := reached $\cup \{e'\}$;
17:         if isTargetState($e'$) then
18:           return (reached, waitlist);
19: return (reached, $\emptyset$);
```

Configurable Predicate Analysis. We use a previous definition of a CPA for predicate abstraction with ABE [12] and configure it. The CPA for predicate analysis $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a stop operator stop, which are defined as follows. (Given a program $P = (L, l_0, G)$, we use X for denoting the set of program variables occurring in P, $\mathcal{P}$ for the set of quantifier-free predicates over variables from X, and $\Pi : L \to 2^{\mathcal{P}}$ for the precision of the predicate abstraction.) Note that this CPA is expected to be used in conjunction with separate CPAs for abstract domains like program counter and call-stack tracking.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is a tuple that consists of a set C of concrete states, a semi-lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$, and a concretization function $[\![\cdot]\!] : E \to C$. The lattice elements $e \in E$ (or abstract states) are tuples $(\psi, l^{\psi}, \varphi) \in (\mathcal{P} \times (L \cup \{l_{\top}\}) \times \mathcal{P})$, where the *state formula* ψ is a boolean combination of predicates that occur in $\Pi(l^{\psi})$, l^{ψ} is the location at which ψ was computed, and φ is a disjunctive *path formula* representing some or all concrete paths from l^{ψ} to the location of state e. The top element of the lattice is the abstract state $\top = (\mathit{true}, l_{\top}, \mathit{true})$. The partial order $\sqsubseteq \subseteq E \times E$ is defined such that for any two elements $e_1 = (\psi_1, l^{\psi}_1, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi}_2, \varphi_2)$ from E the following holds:
$e_1 \sqsubseteq e_2 \Leftrightarrow (e_2 = \top) \vee \big((l^{\psi}_1 = l^{\psi}_2) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2)\big)$
The join operator $\sqcup : E \times E \to E$ yields the least upper bound of the two operands, according to the partial order.

2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ contains all tuples (e, g, e') with $e = (\psi, l^{\psi}, \varphi)$, $e' = (\psi', l^{\psi'}, \varphi')$ and $g = (l, op, l')$ for which the following holds:

$$\begin{cases} (\varphi' = \mathit{true}) \wedge \Big(\psi' = (\mathsf{SP}_{op}(\varphi) \wedge \psi)^{\Pi(l')}\Big) \wedge (l^{\psi'} = l') & \text{if } \mathsf{blk}(e, g) \\ (\varphi' = \mathsf{SP}_{op}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) & \text{otherwise} \end{cases}$$

The 'mode' of the transfer relation, i.e., whether to compute an abstraction, is determined by a block-adjustment operator $\mathsf{blk} : E \times G \to \mathbb{B}$, which is given as parameter to the analysis. Inside each block (the second case) the successor states are created by purely syntactically assembling the strongest post-condition SP of the program code attached to the current edge. At the end of the current block (the first case), an *abstraction state* is created. For such a state, the path formula is reset to true, and the state formula is set to the result of an abstraction computation $(\cdot)^{\Pi(\cdot)}$ using the path formula and previous state formula as input. Thus, the choice of blk determines the block-encoding (i.e., how much to collect in the path formula before abstraction). The precision of the predicate abstraction can vary between program locations (parsimonious precision [9]).

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ for two abstract states $e_1 = (\psi_1, l^{\psi}_1, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi}_2, \varphi_2)$ is defined as follows: $\mathsf{merge}(e_1, e_2) =$

$$\begin{cases} (\psi_2, l^{\psi}_2, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi}_1 = l^{\psi}_2) \\ e_2 & \text{otherwise} \end{cases}$$

This operator combines the two abstract states using a disjunctive path formula, if the abstraction formulas are equal and were computed at the same program location (i.e., if they belong to the same block).

4. The *stop operator* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if e is covered by a state in the set reached: $\mathsf{stop}(e, R) = \exists e' \in R : (e \sqsubseteq e')$

Our refinement procedure first reconstructs all program paths to the error state by traversing the ARG, and creates a concrete path formula for them. This formula is checked for satisfiability using an SMT solver. If it is satisfiable, a feasible error path was found. Otherwise we split the formula, such that one formula is for one block. This way the cut points exactly match the abstraction states in the ARG. We query the solver to produce an inductive Craig interpolant for each cut point.

Configuration / Instantiation. This framework can now be configured to behave similar to the lazy predicate abstraction of BLAST as well as the lazy abstraction with interpolants algorithm of IMPACT by defining the following three items: (1) how the state formula ψ of each abstract state $(\psi, l^{\psi}, \varphi)$ is represented and how the abstraction computation $(\cdot)^{\Pi(\cdot)}$ is defined, (2) how the partial order $\sqsubseteq$ of the lattice is implemented, and (3) how the interpolants are used to modify the ARG during refinement.

The configuration for BLAST-like lazy predicate abstraction works as follows: (1) Each state formula is represented by a binary decision diagram (BDD). It is computed by taking either the cartesian or the boolean abstraction [5] of the conjunction of the previous abstract-state formula and concrete path formula. Which abstraction mechanism is chosen needs to depend on the block size, to achieve a reasonable performance [8]. (2) Coverage checks are done by checking the entailment of the BDDs that represent the state formulas of the two abstraction states for which coverage is checked. (The path formulas of such states are always equal to $true$ and thus need not be considered.) (3) During refinement, the obtained interpolants are split into their basic atoms, and a predicate is created for each of these atoms. All those predicates are added to the precision for the program location for which the interpolant was computed. The first abstraction state in the paths to the error state, for which a new predicate was found, is identified. This state and all states in the ARG that are reachable from it are removed from the ARG (and from the sets reached and waitlist). Its predecessor state is re-added to the waitlist with the new precision. States that were covered by one of the removed states are also re-added to the waitlist.

To configure the framework as IMPACT algorithm, we use the following setup: (1) State formulas are represented by symbolic formulas. All abstraction states have $true$ as their initial state formula, and $\varphi^{\Pi(l)} = true$ for all formulas φ and locations l. (2) Coverage checks are done by querying an SMT solver whether the implication of the state formulas of the two states holds. (3) After the interpolants are computed during refinement, we conjunct them to the state formulas of the abstract states to which they belong. If a state is strengthened (i.e., the interpolants actually added a conjunct to the state formula), we need to re-check all coverage relations of this state. If a previously covered state is now uncovered, we re-add all sink states in the subgraph of the ARG that starts with this state, to the waitlist. We also check each of the strengthened states whether it is now covered by any other state at the same location. If this is successful, we mark the subgraph that starts with that state as covered and remove all leafs therein from the waitlist (we do not need to expand covered states). The only change to the set reached is the removal of all states whose state formula is $false$ and their successors. It is guaranteed that this is the case for the error state (if the error path is infeasible). This refinement procedure is similar to the function REFINE in the original presentation of the IMPACT algorithm [20].

One last configuration option of our framework is the choice of the fcover operator. For the IMPACT algorithm, we may decide to use interpolation-based forced covering. For predicate abstraction, we use an implementation that always returns the set of reached states that was given as parameter (i.e., no change).

This unification makes the essential differences between these two algorithms explicit and removes those differences that have no impact on the performance. We show in our experimental evaluation in Sect. IV that our version of the IMPACT algorithm has similar performance to the original one.

Discussion. Now that we have identified the important differences, we can evaluate them and discuss their meaning. One main difference is that lazy predicate abstraction computes costly abstractions in order to have cheap coverage checks later on. This is an eager technique: computing effort is spent ahead, not knowing whether this will actually pay off. For example, along a long path within a single loop we might compute abstractions for every state, but check coverage only for the states at the loop head. On the other hand, the IMPACT algorithm delays all computation effort until it is actually needed, which means that whenever some information is needed about a state, a costly SMT-solver query is needed.

A further difference is how the coverage relationship is determined. In order to find as much coverage situations as possible and guarantee termination, the IMPACT algorithm may check coverage for a single node several times, specifically whenever it starts expanding nodes in the subtree below this state. Predicate abstraction checks coverage only once directly after the state has been created. However, states are deleted during refinement and might get rediscovered, where they are again checked for coverage.

For predicate abstraction, two choices exist for how to compute an abstraction when creating a state, cartesian abstraction and boolean abstraction. It was shown that if using single-block encoding, boolean abstraction is too slow to be useful and only cartesian abstraction is feasible. However, the latter is imprecise if there are disjunctions in the formulas that represent program operations, because it can infer truth values only for predicates independently from each other. Disjunctions occur, e.g., if pointer-alias information is encoded in the formulas, and thus predicate abstraction with cartesian abstraction may fail to prove properties that rely on this. Boolean abstraction can handle all boolean combinations of predicates and is thus more precise, but is only usable with large blocks. The IMPACT algorithm does not have this problem: it uses the interpolant directly and never loses precision.

Implementation. In order to effectively compare the performance of two algorithms, it is important to implement them in the same tool. Separate tools typically differ in many ways, which have an impact on the performance, for example the programming language, the parser frontend, the used SMT solver, support for additional features like function pointers or pointer aliasing, and optimizations like constant propagation, which are independent from the core algorithm. As a basis for our implementation we took the open-source software verification platform CPACHECKER [11]. It supports verifying C programs, is based on the CPA framework and already has an implementation of lazy predicate abstraction with CEGAR and adjustable-block encoding. We took the existing CPA for predicate abstraction, made the state-formula representation configurable (providing a BDD-based and a symbolic representation with their respective forms of coverage checks), and added an IMPACT-like refinement strategy. We also extended the CPA algorithm to support forced coverings. Thus the implementations of both algorithms differs only in the points listed above; everything else is the same code. The common code includes for example parsing, the traversal algorithm, the encoding of C code into SMT formulas, and the SMT solver.

For comparison, we also implemented the unchanged IMPACT algorithm as described by McMillan [20]. All code that is not related to the algorithm itself and the representation of abstract states is still shared with the other algorithms, so the same parser, formula encoding, and SMT solver are used.

Basic optimizations like caching queries to the SMT solver were implemented in the common code and are thus used by all algorithms. For both versions of the IMPACT algorithm (the original and ours), an optional implementation of the forced-covering optimization was added.

IV. EXPERIMENTAL EVALUATIONS

Benchmark Programs. For our experimental evaluation, we took all 277 C programs from the last competition on software verification [7], out of which 119 programs contain a known specification violation.

Experimental Setup. All experiments were performed on machines with a 3.4 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.04 (64 bit), using Linux 2.6.35 and OpenJDK 1.6. A time limit of 15 minutes and a memory limit of 15 GB were used. We took CPACHECKER from revision 6013 of the 'forced-covering' branch in the repository, and configured it with a Java heap size of 12 GB and MathSAT 4.2.17 as SMT solver. For comparison, we also executed benchmarks with BLAST 2.7 [23] (a tool for lazy predicate abstraction) and WOLVERINE 0.5c [24] (a tool implementing the IMPACT algorithm). For both tools we used the version and the configuration parameters which were submitted to the last software-verification competition. Unfortunately, the original IMPACT tool was not available for benchmarking. We also did benchmarks with UFO 0.1 [2] [3]. For this tool, the programs needed to be pre-processed with a special variant of CIL, compiled with LLVM, and optimized (we ignored the run time necessary for this). This pre-processing failed for 11 benchmarks.

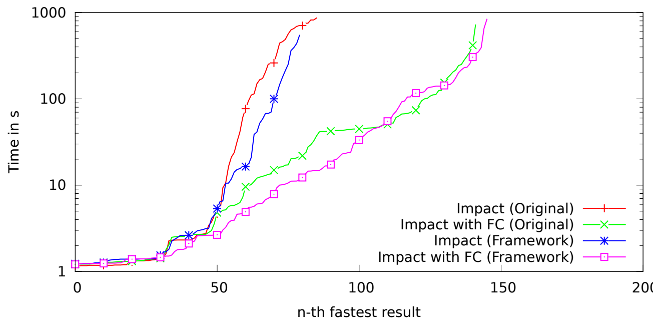


Fig. 1. Original IMPACT algorithm and our framework version; both implemented in CPACHECKER; quantile functions for verification results

Tables with the detailed results, as well as all benchmark programs, the used configurations, scripts, and a ready-to-run version of CPACHECKER are available on the supplementary webpage `http://www.sosy-lab.org/~dbeyer/cpa-uni`.

Variants of the* IMPACT *Algorithm. As a first set of benchmarks, we compare our implementation of the original IMPACT algorithm with the IMPACT algorithm expressed in our unifying framework, both without and with forced covering enabled. The original version is able to solve 86 and 142 instances, respectively, whereas the unifying version is able to solve 80 and 146 instances. The few differences are due to some out-of-memory conditions in the configuration. Figure 1 shows the performance for all successful verification runs of all four configurations using a plot of the quantile functions. The function graph for a configuration yields the maximum run time y (measured as used CPU time) for the xth fastest computed correct results. For example, a time of 10 s for the 100th fastest result would mean that this configuration could successfully verify 100 programs in under 10 s each, and took longer than that for all remaining programs. The x-value for which a graph ends at the top gives the maximal number of successfully verified programs for the configuration. The area below a graph (its integral) represents the accumulated verification time that the configuration needed for all programs that it could verify.

From these results we draw the conclusion that the original version of the IMPACT algorithm and our variant perform

[2]Taken from `http://www.cs.utoronto.ca/~aws/ufo/`

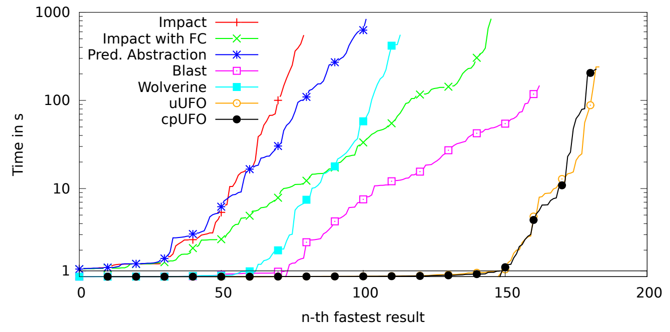


Fig. 2. SBE-based CPAchecker configurations, Blast, Wolverine, and UFO; quantile functions for verification results

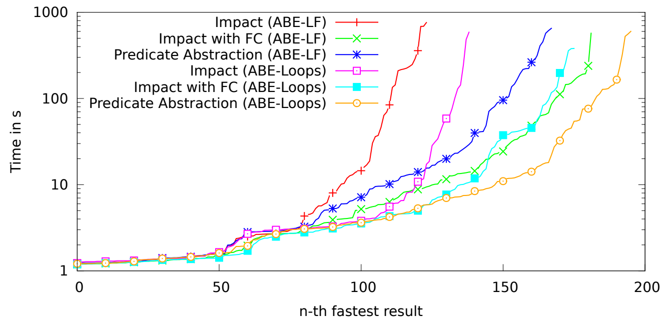


Fig. 3. Large-block CPAchecker configurations to reduce the number of refinements; quantile functions for verification results

TABLE I
Characteristics of different CPAchecker configurations

a) Number of successfully verified programs

	SBE	ABE-LF	ABE-Loops
Impact	80	124	139
Impact with Forced Covering	146	182	176
Predicate Abstraction	102	168	196

b) Average number of refinements for successfully verified programs

	SBE	ABE-LF	ABE-Loops
Impact	513	304	42.5
Impact with Forced Covering	52.2	19.6	14.6
Predicate Abstraction	887	79.5	8.47

similar enough and we are able to further experiment with our unifying framework only.

Benchmarks using Single-Block Encoding. Now we compare the configurations of our analysis framework against each other: the Impact algorithm and lazy predicate abstraction. The former is run with and without the forced-covering optimization. For reference, we also run benchmarks with other tools that implement one of these algorithms: Blast, Wolverine, and UFO. The latter is run in two configurations, without abstraction computations (uUFO, similar to Impact) and with cartesian predicate abstraction (cpUFO). All configurations except the two UFO configurations use single-block encoding, i.e., the former do not group several program statements into larger blocks.

Figure 2 shows the performance of these configurations and tools. The number of solved instances for the CPAchecker configurations can also be seen in the column 'SBE' of Table I a). Comparing the Impact algorithm (1st row) with predicate abstraction (3rd row), we can see that the latter can solve 22 more programs, and is somewhat faster (cf. graph). This indicates that the eagerness of predicate abstraction pays off, and the amount of work spent for computing abstractions is worth the effort. Omitting the abstraction computation and delegating the coverage checks to an SMT solver needs more time, although the SMT solver queries are cached. However, when forced covering is enabled for the Impact algorithm, it can solve 66 more programs, and is much faster than predicate abstraction. These results show that reducing the number of paths in the ARG and the number of refinements is worthwhile, even if substantial effort is needed. One forced covering consists of a satisfiability check and an interpolation query, and can thus be similarly expensive as a refinement. However, the formulas used during a check for forced covering are smaller than those during refinements, and a single successful forced covering can prevent the expansion of a whole subgraph of the ARG and thus save several refinements.

Comparing these results to the results of the other tools that are also shown in the graph is difficult, because the performance characteristics of tools written in Java, OCaml, and C/C++ are typically quite different, different SMT solvers are used, and the amount of work that was put into the tools for adding optimizations and performance tuning differs vastly. This can be seen, for example, by the fact, that in this comparison the lazy predicate abstraction implementations of CPAchecker and Blast have a significant performance difference, although they are conceptually the same.

Benchmarks with Large Blocks. We have already identified that the most important performance factor is the number of refinements, and not the algorithm itself. Thus, we are interested in seeing how both algorithms perform when adjustable-block encoding (ABE) is used to group many program statements into larger blocks. This approach is known to vastly reduce the number of refinements [12]. One important advantage of our unified framework is that we can now use ABE with both algorithms without any further work, although it was originally only designed and implemented for predicate abstraction. Of course, ABE does not save abstraction computations if the Impact algorithm is used, but it still does allow to merge paths and does reduce the number of refinements, and it saves coverage checks as these are only done at block ends. The first results have shown that the original version of Impact behaves similarly to our framework version, therefore, we did not implement adjustable-block encoding for it.

For the larger blocks, we use two different block sizes: ABE-LF (block ends at function entries/exits and loop heads) and ABE-Loops (block ends only at loop heads). The number of solved instances is shown in Table I a) and the performance results in Fig. 3. First of all, the results show the expected improvement in performance and number of solved instances if the block size is increased from SBE to ABE-LF and further to ABE-Loops. This holds for all three configurations with one exception which we will discuss below. The results further confirm that the Impact algorithm without forced covering is the slowest of those configurations regardless of the block

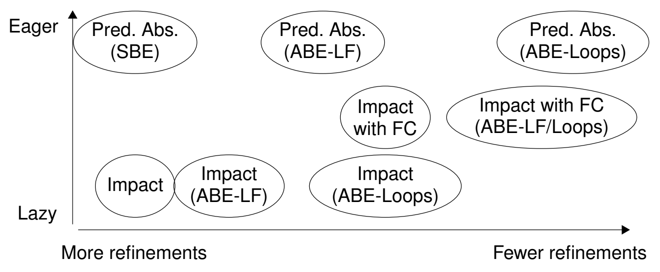


Fig. 4. Classification of various combinations of lazy predicate abstraction and IMPACT with forced covering and adjustable-block encoding

size. For ABE-LF (the medium block size), forced covering provides a performance benefit for the IMPACT algorithm similar to when SBE is used, making it faster than predicate abstraction. However, for ABE-Loops (the largest block size) the results differ: The performance improvement of forced covering still exists, but is smaller than for the other block sizes. The IMPACT algorithm is now slower than predicate abstraction, even with forced covering enabled. Furthermore, this configuration solves fewer instances with ABE-Loops than with ABE-LF, although for all other configurations an increase in the block size also leads to a significant performance increase. The reason for this is that the ABE-Loops block size has reduced the number of refinements already so much that the forced-covering optimization has little chance to achieve a further reduction. Because the only abstraction states that remain belong to loop-head locations, forced covering is attempted only for such states. However, for loop heads the abstract states need to be annotated with loop invariants in order to reach the fixed point, and those invariants are only discovered by refinements, not by forced-covering attempts. The overhead for the unsuccessful attempts then leads to a performance decrease. Table I b) shows the average number of refinements for each successful verification run and confirms this. For SBE and ABE-LF, forced covering can reduce the number of refinements by one order of magnitude, however, for ABE-Loops it only manages to reduce it to one third.

V. CONCLUSION

We have presented a new unifying framework for predicate-based model checking, and expressed the two most successful existing approaches in this framework. This allowed us to gain new insights about these algorithms, especially that the performance benefit of IMPACT compared to SBE-based predicate abstraction is not due to the omitted abstractions, but instead due to the reduction of the number of refinements using forced covering. We can now classify all existing and new configurations as in Fig. 4. We showed that using our framework does not add overhead compared to the original versions of the algorithms. Instead it is beneficial for flexibly experimenting with new configurations, such as combining the IMPACT-based algorithm and adjustable-block encoding.

These experiments confirm that the common property of the most successful configurations is to reduce the number of refinements. The new insights from this experimental study are useful for directing future research on software model checking. Specifically, we have provided an experimental infrastructure to study the impact of the various parameters that distinguish the algorithms.

We plan to extend our framework by incorporating further model-checking algorithms. A comprehensive framework will allow us to learn more about other algorithms and to experiment with new —perhaps even more powerful— strategies for software model checking that were not possible before.

REFERENCES

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proc. TACAS*, LNCS 7214, pages 157–172. Springer, 2012.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. WHALE: An interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI*, LNCS 7148, pages 39–55. Springer, 2012.
[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV*, LNCS 7358, pages 672–678. Springer, 2012.
[4] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *Proc. CAV*, LNCS 6174, pages 119–122. Springer, 2010.
[5] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *Proc. TACAS*, LNCS 2031, pages 268–283. Springer, 2001.
[6] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
[7] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.
[8] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.
[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
[10] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
[11] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
[12] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
[13] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inform.*, 89(4):369–392, 2008.
[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[15] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI*, LNCS 7148, pages 186–201. Springer, 2012.
[16] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
[17] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Proc. POPL*, pages 471–482. ACM, 2010.
[18] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
[20] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.
[21] K. L. McMillan. Lazy annotation for program testing and verification. In *Proc. CAV*, LNCS 6174, pages 104–118. Springer, 2010.
[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.
[23] P. Shved, M. Mandrykin, and V. Mutilin. Predicate analysis with BLAST 2.7. In *Proc. TACAS*, pages 525–527. Springer, 2012.
[24] G. Weissenbacher, D. Kröning, and S. Malik. WOLVERINE: Battling bugs with interpolants. In *Proc. TACAS*, pages 556–558. Springer, 2012.

Incremental Upgrade Checking by Means of Interpolation-based Function Summaries

Ondrej Sery*† Grigory Fedyukovich* Natasha Sharygina*
*Formal Verification Lab, University of Lugano, Switzerland
†D3S, Faculty of Mathematics and Physics, Charles University, Czech Rep.

Abstract—**During its evolution, a typical software/hardware design undergoes a myriad of small changes. However, it is extremely costly to verify each new version from scratch. As a remedy to this problem, we propose to use function summaries to enable incremental verification of the evolving systems. During the evolution, our approach maintains function summaries derived using Craig's interpolation. For each new version, these summaries are used to perform a local incremental check. Benefit of this approach is that the cost of the check depends on the extent of the change between the two versions and can be performed cheaply for incremental changes without resorting to re-verification of the entire system. Our implementation and experimentation in the context of the bounded model checking for C confirms that incremental changes can be verified efficiently for different classes of industrial programs.**

I. Introduction

Software and hardware designs are usually not written all at once, but are built incrementally, due to numerous reasons: 1) requirements change and have impact on the design and implementation; 2) errors are often discovered late in the design cycle and must be removed; 3) software components are updated or substituted to adapt to architectural and requirement changes; just to name a few. Changes are done frequently during the lifetime of many products and can introduce errors that were not present in the old versions, or expose errors that were present before but did not get exposed. The state of the affairs is that the correctness of the system has to be re-validated from scratch after any (even minor) change. Often the cost of this validation dominates costs of the products.

Currently, re-validation mostly relies on the execution of extensive test suits, which is inherently not exhaustive; fault localization is mainly manual and driven by experts' knowledge of the system; fault fixing often introduces new faults that are hard to detect and remove. To address this problem, this paper presents a new fully automated approach that extends formal verification by model checking to the problem of validation of system upgrades. The new technique focuses on the incremental changes and takes advantage of the effort already invested in the verification of previous versions. The target of our approach is to avoid (when possible) re-validation of the new system and to reduce analysis only to the parts of the system which were affected by the change.

The advantages of model checking are often shaded by its high consumption of computational resources (known as the state-space explosion problem). Many efficient complexity reduction algorithms have been developed to cope with this problem among which the representative approaches are symbolic verification such as Bounded Model Checking (BMC) [1], and different types of automated abstraction (predicate abstraction [2], interpolation-based reasoning [3], function summarization [4], [5], [6], [7], etc.). Most state-of-the-art model checking tools implement some (or combinations) of these methods in order to deal with complex designs. Notably, combinations of such techniques are known to be crucial for combating the high complexity of verification.

This paper presents a solution to the upgrade checking problem that extends the existing efficient techniques known to work well for standalone verification to the problem of analysis of system changes. In particular, it presents an incremental bounded model checking approach that uses function summarizations for local upgrade checks. The upgrade checking algorithm maintains program function summaries (i.e., over-approximations of the actual behavior of the functions, in our case computed by means of Craig interpolation [7]) and when a new version arrives, it checks if the summaries of the modified functions are still valid over-approximations. This is a local and cheap check. If it succeeds, the upgrade is safe with respect to both the preserved and newly added behaviors. If not, the check is propagated by the call tree traversal to the caller of the modified function. As soon as the safety is established, new summaries are generated using Craig interpolation for all the functions with invalid summaries. If the check fails for the call tree root (the main function of the program), an error trace is created and reported to the user as a witness to the violation.

The upgrade checking algorithm implements the refinement strategy for dealing with spurious behaviors which can be introduced during computation of the over-approximated summaries. The refinement procedure for upgrade checks builds on ideas of using various summary substitution scenarios [7], [8] and extends it to 1) handle summaries of nested function calls and 2) consequently to use them to further simplify the validity checks of the upgraded functions summaries. Failures of such checks may be due to the use of too weak summaries, in which case, the refinement is used to expand the involved function calls on demand.

We developed a prototype implementation of the proposed algorithm and evaluated it using a set of industrial benchmarks. Our experimentation confirms that the incremental analysis of

This work is partially supported by the European Community under the call FP7-ICT-2009-5 — project PINCETTE 257647.

upgrades containing incremental changes is often orders of magnitude faster than analysis performed from scratch.

Although we implemented the proposed upgrade checking algorithm in the context of bounded model checking, the algorithm itself is more general and can be employed in other contexts, where over-approximative function summaries are used. For example, the WHALE approach [6] designed for standalone verification could be easily extended to incremental upgrade checking using our algorithm.

In summary, the contributions of the paper are as follows:

- It presents a fully automated model-checking-based technique for verification of incremental upgrades. It is able to re-validate all previously established safety properties and to detect newly introduced errors.
- It efficiently combines bounded model checking with function summarization for *local* and *incremental* analysis of changes. The use of Craig interpolation to compute summaries allows capturing symbolically all execution traces through the function and, together, with the local per-function checks of the new algorithm, results in the efficient analysis procedure.
- It reports on the prototype implementation of the new technique and its validation on industrial benchmarks.

The rest of the paper is organized as follow. Sect. II defines the notation and presents background on function summarization in BMC. Sect. III presents the new upgrade checking algorithm and proves its correctness. Sect. IV describes implementation and evaluation of the approach. Sect. V discusses the related work and Sect. VI concludes the paper.

II. Background and Previous Work

Craig Interpolation [9] is a popular abstraction technique widely used in Model Checking. Given a pair of formulas (A, B), *Craig interpolant* of (A, B) is a formula I such that $A \to I$, $I \wedge B$ is unsatisfiable, and I contains only free variables common to A and B. For an unsatisfiable pair of formulas (A, B), an interpolant always exists [9]. As shown in [10], an interpolant can be constructed from a proof of unsatisfiability by an algorithm referred as *Pudlák's algorithm*. Although other algorithms exist, we will focus on Pudlák's throughout the paper. Interpolants are useful in various verification gambits including refinement of predicate abstraction [4], and bounded model checking [3] to name a few. The following outlines how interpolation is used for function summarization in BMC [7].

BMC is aimed at searching for errors in a program within the given number (bound) of loop iterations and recursion depth. First, it unwinds the program according to the bound. Second, it constructs the Static Single Assignment (SSA) form of the program, supplies it with the negated property to be checked, and encodes it into a logical formula, a *BMC formula*. The formula is satisfiable if and only if an error is reachable in the unwound program. If the formula is satisfiable, a satisfying assignment identifies a trace leading to an error. If unsatisfiable, the program is safe.

Standard BMC constructs a monolithic BMC formula with all function calls inlined. To make interpolation applicable for extraction of function summaries, we construct BMC formula so that each function call is represented by a separate conjunct, and call it a *partitioned BMC* (PBMC) formula. To describe construction of PBMC formula in more details, we use the notion of an unwound program in terms of its call tree.

An *unwound program* for a bound ν is a tuple $P_\nu = (F, f_{main})$, s.t. F is a finite set of functions, $f_{main} \in F$ is an entry point and every loop and recursive call is unrolled (unwound) ν times. In addition, we define a relation *child* $\subseteq F \times F$ which relates each function f to all the functions invoked by f. Relation *subtree* $\subseteq F \times F$ is a transitive closure of *child*. $\hat{F}$ denotes the finite set of unique function calls, with $\hat{f}_{main}$ being the implicit call to the program entry point. The relations *child* and *subtree* are naturally extended to $\hat{F}$, s.t. $\forall \hat{f}, \hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \to child(f, g)$, and *subtree* is a transitive closure of the extended relation *child*. A *summary* of a function is a relation over its input and output variables, which over-approximates the precise behavior of the given function. This means that a summary contains all possible behaviors of the function (under the given bound ν) and possibly more. We use $\mathbb{S}$ to denote the set of all summaries.

Algorithm 1 summarizes the method for construction of function summaries in BMC. There are two major differences from the standard BMC algorithm that should be pointed out. First, the PBMC formula is constructed as a conjunction of parts representing individual functions. Second, function summaries are extracted using interpolation for every individual part of the PBMC formula.

PBMC formula construction (line 1). The PBMC formula is constructed in the recursive method `CreateFormula` as follows.

$$\texttt{CreateFormula}(\hat{f}) \triangleq \phi_{\hat{f}} \wedge \bigwedge_{\hat{g} \in \hat{F} : child(\hat{f}, \hat{g})} \texttt{CreateFormula}(\hat{g})$$

For a function call $\hat{f} \in \hat{F}$, the formula is constructed by conjunction of the partition $\phi_{\hat{f}}$ reflecting the body of the function and a separate partition for every nested function call. The logical formula $\phi_{\hat{f}}$ is constructed from the SSA form of the body of the function f. The bodies of the nested calls are encoded into separate logical formulas (using a recursive call to `CreateFormula`) and thus separate partitions in the resulting PBMC formula. In addition, $\phi_{\hat{f}}$ contains special propositional symbols to bind the individual partitions together. An example of such a symbol is $error_{\hat{f}}$, which is constrained to be true if and only if the function call $\hat{f}$ results in an error. Consequently, $error_{\hat{f}_{main}}$ encodes reachability of an error in the entire program (for further details see [7]).

Summarization (line 6). If the PBMC formula is unsatisfiable, i.e., the program is safe, the algorithm proceeds with interpolation. The function summaries are constructed as interpolants from a proof of unsatisfiability of the PBMC formula. In order to generate the interpolant, for each function call $\hat{f}$ the PBMC formula is split into two parts. First, $\phi_{\hat{f}}^{subtree}$

Algorithm 1: Function summarization in BMC [7]

```
Input: Unwound program $P_\nu = (F, f_{main})$ with function calls $\hat{F}$
Output: Verification result: {SAFE, UNSAFE}, mapping of function calls to their summaries $summaries$
Data: $\phi$: PBMC formula
1 $\phi \leftarrow$ CreateFormula($\hat{f}_{main}$) $\wedge$ $error_{\hat{f}_{main}}$;
2 $result, proof \leftarrow$ Solve($\phi$) ;   // run SAT-solver
3 if $result$ = SAT then
4     return UNSAFE;
5 foreach $\hat{f} \in \hat{F}$ do          // extract summaries
6     $summaries(\hat{f}) \leftarrow$ Interpolate($proof, \hat{f}$);
7 end
8 return SAFE;
```

corresponds to the partitions representing the function call $\hat{f}$ and all the nested functions. Second, $\phi_{\hat{f}}^{env}$ corresponds to the context of the call $\hat{f}$, i.e., to the rest of the encoded program.

$$\phi_{\hat{f}}^{subtree} \triangleq \bigwedge_{\hat{g}\in\hat{F}:subtree(\hat{f},\hat{g})} \phi_{\hat{g}}$$

$$\phi_{\hat{f}}^{env} \triangleq error_{\hat{f}_{main}} \wedge \bigwedge_{\hat{h}\in\hat{F}:\neg subtree(\hat{f},\hat{h})} \phi_{\hat{h}}$$

Therefore, for each function call $\hat{f}$, the `Interpolate` method separates the PBMC formula into $A \equiv \phi_{\hat{f}}^{subtree}$ and $B \equiv \phi_{\hat{f}}^{env}$ and generates an interpolant $I_{\hat{f}}$ for the pair (A, B). Such interpolant $I_{\hat{f}}$ is a summary for the function f. The generated interpolants are associated with the function calls by a mapping[1] *summaries*: $\hat{F} \rightarrow \mathbb{S}$, i.e., *summaries*$(\hat{f}) = I_{\hat{f}}$.

Refinement. When the same program is being verified again (e.g., with respect to a different property), the exact function calls can be substituted by the constructed summaries. In this case, the method `CreateFormula` of Algorithm 1 is replaced by the following:

$$\texttt{CreateFormula}(\hat{f}) \triangleq \phi_{\hat{f}} \wedge \Big(\bigwedge_{\hat{g}\in\hat{F}:child(\hat{f},\hat{g})\wedge\Omega(\hat{g})=inline} \texttt{CreateFormula}(\hat{g})\Big) \Big(\bigwedge_{\hat{g}\in\hat{F}:child(\hat{f},\hat{g})\wedge\Omega(\hat{g})=sum} summaries(\hat{g})\Big)$$

where a *substitution scenario* $\Omega : \hat{F} \rightarrow \{inline, sum, havoc\}$ determines how each function call should be handled. Initially, Ω depends on existence of function summaries. If a summary of a function exists, it is used to represent the function - *sum*. If not, the function is either represented precisely - *inline* (*eager* scenario), or abstracted away - *havoc* (*lazy* scenario).

If the resulting formula is satisfiable, it may be due to too coarse summaries. Refinement, first, identifies which summaries affect satisfiability of the PBMC formula. This is done by analyzing the occurrence of summaries along an error trace, determined by a satisfying assignment and by dependency analysis. Second, the *refined* substitution scenario Ω' is constructed from Ω by mapping the function calls corresponding to the identified summaries to *inline*. Then, the next iteration of the algorithm is run using Ω'. If no summary is identified for refinement, the error is real.

[1]Here, we consider only a single summary per a function call for the sake of simplicity. This still means multiple summaries per a single function called multiple times. Our prototype implementation does not have this restriction.

III. Upgrade Checking

This section describes our solution to the upgrade checking problem, the incremental summary-based model checking algorithm. As an input, the algorithm takes two versions of the system, old and new, and the function summaries of the old version. If the old version or its function summaries are not available (e.g., for the initial version of the system), a bootstrapping verification run is needed to analyze the entire new version of the system and to generate the summaries, which are then maintained during the incremental runs.

The incremental upgrade check is performed in two phases. First, in the preprocessing phase, the two versions are compared at the syntactical level. This allows identification of functions that were modified (or added) and which summaries need rechecking (or they even do not exist yet). An additional output of this phase is an updated mapping *summaries*, which maps function calls in the new version to the old summaries.

For example, Figure 1-a depicts an output of the preprocessing, i.e., a call tree of a new version with two changed function calls (gray fill). Their summaries need rechecking. In this case, all function calls are mapped to the corresponding old summaries (i.e., functions were possibly removed or modified, but not added). Summaries of all the function calls marked by a question mark may yet be found invalid. Although the code of the corresponding functions may be unchanged, some of their descendant functions were changed and may eventually lead to invalidation of the ancestor's summary.

In the second phase, the actual upgrade check is performed. Starting from the bottom of the call tree, summaries of all functions marked as changed are rechecked. That is, a cheap local check is performed to show that the corresponding summary is still a valid over-approximation of the function's behavior. If successful, the summary is still valid and the change (i.e., rightmost node in Figure 1-b) does not affect correctness of the new version. If the check fails, the summary is invalid for the new version and the check needs to be propagated to the caller, towards the root of the call tree (Figure 1-b,c). When the check fails for the root of the call tree (i.e., program entry point $\hat{f}_{main}$), a real error is identified and reported to the user. The following first presents this basic algorithm in more details and then describes its optimization with a refinement loop and proves its correctness. Note that we will describe the upgrade checking algorithm instantiated in the context of bounded model checking. However, the algorithm is more general and can be applied in other approaches relying on over-approximative function summaries.

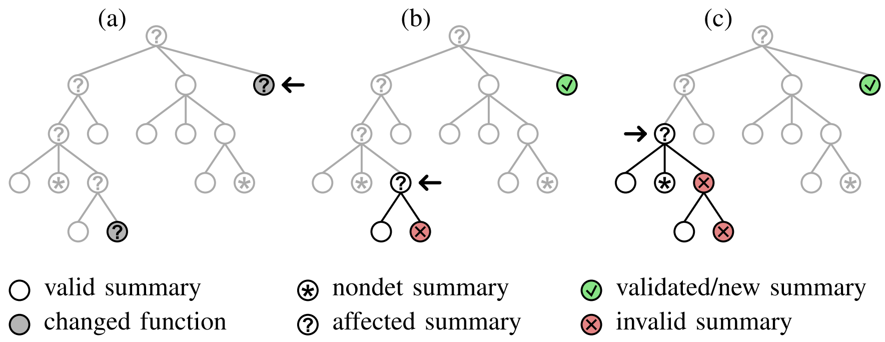


Figure 1: Progress of the upgrade checking algorithm; the faded parts of the call tree were not yet analyzed by the algorithm

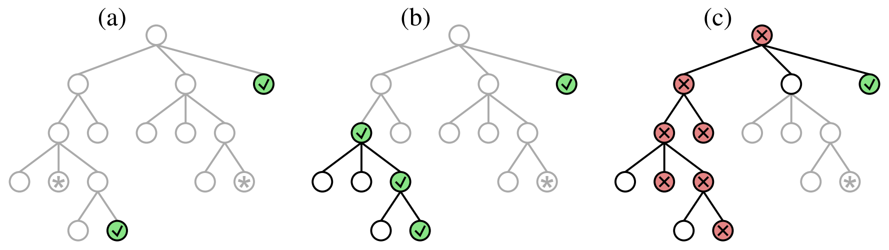


Figure 2: Sample outcomes of Alg. 2; analyzing the faded parts of call tree is not required to decide safety of the upgrade

A. *Basic Algorithm*

We proceed by presenting the basic upgrade checking algorithm (Alg. 2). As an input, Alg. 2 takes the unwound new version of the program, a mapping *summaries* from the function calls in the new version to the summaries from the old version, and a set *changed* marking the function calls corresponding to the functions that were changed or added in the new version (as an output of the preprocessing).

The algorithm keeps a set D of function calls that require rechecking. Initially, this set contains all the function calls marked by *changed* (line 1). Then the algorithm repeatedly removes a function call $\hat{f}$ from D and attempts to check validity of the corresponding summary in the new version. The algorithm picks $\hat{f}$ so that no function call in the subtree of $\hat{f}$ occurs in D (line 3). This ensures that summaries in the subtree of $\hat{f}$ were already analyzed (shown valid or invalid).

The actual summary validity check occurs on lines 6, 7. First, the PBMC formula encoding the subtree of $\hat{f}$ is constructed and stored as ϕ. Then, conjunction of ϕ with a negated summary of $\hat{f}$ is passed to a solver for the satisfiability check. If unsatisfiable, the summary is still a valid over-approximation of the function's behavior. Here, the algorithm obtains a proof of unsatisfiability which is used later to create new summaries to replace the invalid or missing ones (line 9-11). If satisfiable, there is a combination of inputs and outputs of the function f that is not covered by its original summary, thus the summary is not valid for the new version (line 14). In this case, either a real error is identified (lines 16, 17) or the check is propagated to the function caller (line 18).

Note that if the chosen function call $\hat{f}$ has no summary, e.g., due to being a newly added function, the check is propagated to the caller immediately (condition at line 5) and the summary of $\hat{f}$ is created later when the check succeeds for an ancestor function call.

The algorithm always terminates with either SAFE or UNSAFE value. Creation of each PBMC formula terminates because they operate on the already unwound program. The algorithm terminates with SAFE result (line 20) when all function calls requiring rechecking were analyzed (line 2). Either all the summaries possibly affected by the program change are immediately shown to be still valid over-approximations (see Figure 2-a) or some are invalid but the propagation stops at a certain level of the call tree and new valid summaries are generated (see Figure 2-b). The algorithm terminates with UNSAFE result (lines 17), when the check propagates to the call tree root, $\hat{f}_{main}$, and fails (see Figure 2-c). In this case, a real error is encountered and reported to the user.

B. *Optimization and Refinement*

To optimize the upgrade check, old function summaries can be used to abstract away the function calls. Consider the validity check of a summary of a function call $\hat{f}$. Suppose there exists a function call $\hat{g}$ in the subtree of $\hat{f}$ together with its old summary, already shown valid. Then this summary can be substituted for $\hat{g}$, while constructing the PBMC formula of $\hat{f}$ (line 6). This way, only a part of the subtree of $\hat{f}$ needs to be traversed and the PBMC formula ϕ can be substantially smaller compared to the encoding of the entire subtree.

If the resulting formula is SAT, it can be either due to a real violation of the summary being checked or due to too coarse summaries used to substitute some of the nested function calls. In our upgrade checking algorithm, this is handled in similar way as in the refinement of the standalone

Algorithm 2: Upgrade checking algorithm

```
Input: Unwound program $P_\nu = (F, f_{main})$ with function
       calls $\hat{F}$, mapping $summaries : \hat{F} \to \mathbb{S}$, set
       $changed \subseteq \hat{F}$
Output: Verification result: {SAFE, UNSAFE}
Data: $D \subseteq \hat{F}$: function calls to recheck, $\phi$: PBMC
      formula, $invalid \subseteq \mathbb{S}$: set of invalid summaries
1  $D \leftarrow \{\hat{f} \mid \hat{f} \in changed\}$, $invalid \leftarrow \emptyset$;
2  while $D \neq \emptyset$ do
3    choose $\hat{f} \in D$, s.t. $\forall \hat{h} \in D : \neg subtree(\hat{f}, \hat{h})$;
4    $D \leftarrow D \setminus \{\hat{f}\}$;
5    if $\hat{f} \in dom(summaries)$ then
6      $\phi \leftarrow$ CreateFormula($\hat{f}$);
7      $result, proof \leftarrow$ Solve($\phi \wedge \neg summaries(\hat{f})$);
8      if $result =$ UNSAT then
9        for $\hat{g} \in \hat{F} : subtree(\hat{f}, \hat{g}) \wedge (\hat{g} \notin$
         $dom(summaries) \vee summaries(\hat{g}) \in invalid)$
         do
10         $summaries(\hat{g}) \leftarrow$ Interpolate($proof, \hat{g}$);
11       end
12       continue;
13     end
14     $invalid \leftarrow invalid \cup \{summaries(\hat{f})\}$;
15   end
16   if $\hat{f} = \hat{f}_{main}$ then
17     return UNSAFE;          // real error found
18   $D \leftarrow D \cup \{parent(\hat{f})\}$;      // check parent
19 end
20 return SAFE;               // system is safe
```

verification by analyzing the satisfying assignment. The set of summaries used along the counter-example is identified. Then it is further restricted by dependency analysis to only those possibly affecting the validity. Every summary in the set is marked as *inline* in the next iteration. If the set is empty, the check fails and the summary is shown invalid. This refinement loop (replacing lines 6, 7 in Alg. 2) iterates until validity of the summary is decided.

This optimization does not affect termination of the algorithm (in each step at least one of the summaries is refined). Regarding complexity, in the worst case scenario, i.e. when a major change occurs, the entire subtree is refined one summary at a time for each node of the call tree. This may result in a number of solver calls quadratic in the size of the call tree, where the last call is as complex as the verification of the entire program from scratch. This paper focuses on incremental changes and thus for most cases there is no need for the complete call graph traversal. Moreover, the quadratic number of calls can be easily mitigated by limiting the refinement laziness using a threshold on the number of refinement steps and disabling this optimization when the threshold is exceeded.

C. Correctness

This section demonstrates the correctness of the upgrade checking algorithm, i.e., given an unwinding bound ν, the algorithm always terminates with the correct answer w.r.t. ν. Note that throughout this section, program safety is understood considering the bound ν[2]. Also, we use $\sigma_{\hat{f}}$ as a shortcut for *summaries*$(\hat{f})$. The key insight for the correctness is that after each successful run of Alg. 2 (i.e., when *SAFE* is returned), the following two properties are maintained.

$$error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \to \bot \qquad (1)$$

Given each function call $\hat{f}$ and its children calls $\hat{g}_1, \ldots, \hat{g}_n$:

$$\sigma_{\hat{g}_1} \wedge \ldots \wedge \sigma_{\hat{g}_n} \wedge \phi_{\hat{f}} \to \sigma_{\hat{f}} \qquad (2)$$

Theorem 1 is needed to prove the correctness of Alg. 2. It considers properties of interpolants (a.k.a. *tree interpolants*) generated from the same resolution proof using Pudlák's algorithm (we kindly refer reader to [10] for details).

Theorem 1. *Let $X_1 \wedge \ldots \wedge X_n \wedge Y \wedge Z$ be an unsatisfiable formula and let $I_{X_1}, \ldots, I_{X_n}$, and I_{XY} be Craig interpolants for pairs $(X_1, X_2 \wedge \ldots \wedge X_n \wedge Y \wedge Z)$, …, $(X_n, X_1 \wedge \ldots \wedge X_{n-1} \wedge Y \wedge Z)$, and $(X_1 \wedge \ldots \wedge X_n \wedge Y, Z)$ respectively, derived using Pudlák's algorithm over a resolution proof $\mathbb{P}$. Then $(I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge Y) \to I_{XY}$.*

We first state and prove a version of Theorem 1 limited to two partitions abstracted by interpolants, then we generalize.

Lemma 1. *Let $X \wedge Y \wedge Z$ be an unsatisfiable formula and let I_X, I_Y, and I_{XY} be Craig interpolants for pairs $(X, Y \wedge Z)$, $(Y, X \wedge Z)$, and $(X \wedge Y, Z)$ respectively, derived using Pudlák's algorithm over a resolution proof $\mathbb{P}$. Then $(I_X \wedge I_Y) \to I_{XY}$.*

Proof: By structural induction over the resolution proof, we show that $(I_X \wedge I_Y) \to I_{XY}$ for all partial interpolants at all nodes of the proof $\mathbb{P}$. As a base case, there is a clause C and we need to consider three cases: $C \in X$, $C \in Y$, and $C \in Z$. When $C \in X$, we have $(false \wedge true) \to false$, which holds. The case $C \in Y$ is symmetric. When $C \in Z$, we have $(true \wedge true) \to true$, which again obviously holds.

As an inductive step, we have a node $C_1 \vee C_2$ representing resolution over a variable x with parent nodes $x \vee C_1$ and $\neg x \vee C_2$. From the inductive hypothesis, we have partial interpolants I^1_X, I^1_Y, and I^1_{XY} for the node $x \vee C_1$ so that $(I^1_X \wedge I^1_Y) \to I^1_{XY}$ and partial interpolants I^2_X, I^2_Y, and I^2_{XY} for the node $\neg x \vee C_2$ so that $(I^2_X \wedge I^2_Y) \to I^2_{XY}$. We need to consider the different cases of coloring of x based on its occurrence in different subsets of the parts of the formula $X \wedge Y \wedge Z$. The cases are summarized in Table I. In case $x \in X$, we have:

$$I_X \equiv I^1_X \vee I^2_X, \quad I_Y \equiv I^1_Y \wedge I^2_Y$$
$$I_{XY} \equiv I^1_{XY} \vee I^2_{XY}$$

[2] We expect the same ν for the old and new version. To keep correctness, if the user increases the bound for a specific loop, the corresponding function has to be handled as if modified.

Table I: Variable classes; a, b: x occurs only in A, resp. B, ab: x occurs in both A and B

x in	class of x for partial interpolant		
	I_X	I_Y	I_{XY}
X	a	b	a
Y	b	a	a
Z	b	b	b
$X+Y$	ab	ab	a
$X+Z$	ab	b	ab
$Y+Z$	b	ab	ab
$X+Y+Z$	ab	ab	ab

Using the inductive hypothesis, we have $((I_X^1 \vee I_X^2) \wedge I_Y^1 \wedge I_Y^2) \rightarrow (I_{XY}^1 \vee I_{XY}^2)$, which is the required claim $(I_X \wedge I_Y) \rightarrow I_{XY}$. The case $x \in Y$ is symmetric.

In case $x \in Z$, we have:

$$I_X \equiv I_X^1 \wedge I_X^2, \quad I_Y \equiv I_Y^1 \wedge I_Y^2$$
$$I_{XY} \equiv I_{XY}^1 \wedge I_{XY}^2$$

Using the inductive hypothesis, we have $(I_X^1 \wedge I_X^2 \wedge I_Y^1 \wedge I_Y^2) \rightarrow (I_{XY}^1 \wedge I_{XY}^2)$, which is the required claim $(I_X \wedge I_Y) \rightarrow I_{XY}$.

In case $x \in X+Y+Z$, using $sel(x, S, T)$ as a shortcut for $(x \vee S) \wedge (\neg x \vee T)$, we get:

$$I_X \equiv sel(x, I_X^1, I_X^2), \quad I_Y \equiv sel(x, I_Y^1, I_Y^2)$$
$$I_{XY} \equiv sel(x, I_{XY}^1, I_{XY}^2)$$

Using the inductive hypothesis and considering both possible values of x, we have $(sel(x, I_X^1, I_X^2) \wedge sel(x, I_Y^1, I_Y^2)) \rightarrow sel(x, I_{XY}^1, I_{XY}^2)$, which is the required claim $(I_X \wedge I_Y) \rightarrow I_{XY}$. The other cases where $x \in X+Y$ or $x \in X+Z$ or $x \in Y+Z$ are subsumed by this case as $(P \wedge Q) \rightarrow sel(x, P, Q) \rightarrow (P \vee Q)$. Structural induction yields $(I_X \wedge I_Y) \rightarrow I_{XY}$ for the root of the proof tree and for the final interpolants. ■

When we apply the result of Lemma 1 iteratively, we obtain a generalized form for cases using multiple interpolants mixed with original parts of the formula, i.e., a proof of Theorem 1.

Proof: By iterative application of Lemma 1, we get $(I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge I_Y) \rightarrow I_{XY}$, where $I_Y$is Craig interpolant for the pair $(Y, X_1 \wedge \ldots \wedge X_n \wedge Z)$ derived using Pudlák's algorithm over the resolution proof $\mathbb{P}$. Using $Y \rightarrow I_Y$, we obtain the claim $(I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge Y) \rightarrow I_{XY}$. ■

In the following two lemmas, we first show that the properties (1, 2) hold after an initial whole program check. Then we show that the properties are maintained between the individual successful upgrade checks.

Lemma 2. *After an initial whole-program check, the properties (1, 2) hold over the call tree annotated by the generated interpolants.*

Proof: Recall that the summaries are constructed only when the program is safe. In other words, $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$. Thus, by definition of interpolation, $error_{\hat{f}_{main}} \wedge I_{\hat{f}_{main}}$ is obviously unsatisfiable, i.e., the property (1) holds. The property (2) follows from Theorem 1. It suffices to choose $X_i \equiv \phi^{subtree}_{\hat{g}_i}$ for $i \in 1..n$, $Y \equiv \phi_{\hat{f}}$, and $Z \equiv \phi^{env}_{\hat{f}}$. ■

Lemma 3. *The properties (1, 2) are reestablished whenever the upgrade checking algorithm successfully finishes (SAFE is returned).*

Proof: The property (1) could be affected only when the summary of $\hat{f}_{main}$ is recomputed (line 10). However, this happens only when we are checking the root of the tree and, at the same time, the check succeeds (line 8). Therefore, by definition of interpolation, the property (1) is maintained.

If Alg. 2 successfully finishes, then each function call $\hat{f}$ with an invalidated summary must have been assigned a new summary $\sigma_{\hat{f}}$ (line 10) when some of its ancestors $\hat{h}$ passed the summary validity check (line 8). Otherwise, the invalidation would propagate to the root of the call tree and eventually produce an *UNSAFE* result. Therefore, it suffices to show that the newly generated interpolants satisfy the property (2). For this purpose, we can use the same argument as in the proof of Lemma 2, again relying on Theorem 1. Note that if any already valid summaries are used in the summary validity check, we keep those (see condition on line 9) instead of generating new ones. This is sound as we know that $\sigma_{\hat{g}_i} \rightarrow I_{X_i}$, which is consistent with our claim. Analogically, we also keep the old summary $\sigma_{\hat{h}}$ for the root of the subtree that passed the check and caused generation of the new summaries. This is sound as $I_{\hat{h}} \rightarrow \sigma_{\hat{h}}$ is implied by the summary validity check. ■

We now show that the properties (1, 2) are strong enough to show that the entire program is safe.

Theorem 2. *When the program call tree annotated by interpolants satisfies the properties (1, 2), then* error$_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$ *(i.e., the entire program is safe).*

Proof: The property (1) yields $error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \rightarrow \bot$. Repeated application of the property (2) to substitute all interpolants on the right hand side yields the claim $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$. ■

We proved correctness of the upgrade checking algorithm in the context of bounded model checking and interpolation-based function summaries. The upgrade checking algorithm, however, is not bound to this context and can be employed also in other verification approaches based on over-approximative function summaries (including the use of other interpolation algorithms). The key ingredient of the correctness proof, the property (2), has to be ensured for the particular application.

IV. Evaluation

We implemented a prototype, eVolCheck, of the upgrade checking algorithm for incremental verification. It performs the checks of upgrades using outputs of the previous check and provides its own outputs to the next one. The required input is function summaries of the previous version. eVolCheck communicates with FunFrog [7] for bootstrapping (to create function summaries of the original code) and exploits its interface with the OpenSMT solver [11] to solve a PBMC formula, encoded propositionally, and to generate interpolants. Altogether, the tool implements two major tasks: syntactic difference check, and the actual upgrade check.

For the first task, we implemented a syntactic difference tool called `goto-diff`. First, `goto-diff` extracts intermediate representations of a pair of (old and new) programs expressed in simple statements (assignments, guards, gotos, function calls) and constructs a so called `goto-binary`. For this, we use the `goto-cc`[3] verification front-end. Since, `goto-binary` is a semantically clean representation of the source code, some syntactically different programs may result in an equivalent representation, i.e., some refactoring changes may be shown safe already at this stage without running the upgrade checking algorithm. Second, `goto-diff` compares the call trees of the programs. For each pair of matching functions, `goto-diff` analyzes their bodies.[4] Unreachable functions of the programs are not processed. Finally, `goto-diff` outputs the new call tree, marked by old summaries and the *changed* set of modified functions. Afterwards, eVolCheck performs the actual upgrade check by following the steps of Alg. 2. After its completion, the result of the change validation is returned to the user. If the upgrade is unsafe, an error is reported and the user is expected to fix it. When the fix is done, it is checked against the latest correct version. Otherwise, the program is correct, the new call tree and the summaries are stored for the use by the next upgrade checking run.

Experiments. We evaluated eVolCheck on a set of industrial benchmarks. Four of them (`VTT_`n) were provided by our industrial partner, the VTT company. The rest is derived from a library of Windows device drivers (`floppy_`n, `kbfiltr_`n, `diskperf_`n). We invented all changes artificially.

Safety of all benchmarks was verified against assertions, either existing in the code or inserted by us into code without assertions. Table II contains results of the experiments. Each row corresponds to a different benchmark, groups of columns represent statistics about the bootstrapping verification and verification of two upgrades, respectively. NoI estimates the size of the original source code as a number of instructions in the `goto-binary` (NoI is an accurate representation of code without definitions, and often represents much higher number of lines of code). IC represents the number of changed instructions between current and the previous version. The overhead introduced by upgrade checking, i.e. the syntactic difference check (Diff) and the interpolants generation (Itp), is also included in the total running time (Total). To show advantages of our upgrade checking approach, for each change we calculated the speedup (Speedup) of the upgrade check versus verification of the changed code from scratch, performed only for the sake of comparison reasons and hidden from the table.

In order to demonstrate different performance of our technique, we chose two different classes of changes for each benchmark. The first class (1st change) represents changes with small impact. As expected, those can be verified with a few local checks. The second one (2nd change) presents upgrades that affect large portion of the code, potentially causing traversal of the complete call tree of the program.

[3]http://www.cprover.org/goto-cc/

[4]Two functions match iff their signatures are the same (function name, types and order of arguments, and return type).

Our experiments demonstrate that for the class of problem with small impact, the upgrade checking approach outperforms the standalone verification (order(s) of magnitude speedup). For the second class of changes, the performance of the upgrade check varies. For some cases, analysis could be done locally and the speedup is still substantial. For cases where the algorithms needed to analyze large portion of the call tree, the performance, as expected, degrades. Note that the bad performance occurs when the change introduces a bug (indicated by '—' in the Itp column; the PBMC formula is satisfiable and interpolants are not generated). In this case, the upgrade check traverses to the root of the call tree, in order to reconstruct a complete error trace. Of course, this can be an easy task when the change is close to the root of the call tree (e.g., in the `floppy_D` benchmark). The results support our initial intuition that upgrade checking works well for incremental changes, which is the most common class in the evolution of systems.

V. Related Work

The area of software upgrade checking is not as studied as model checking of standalone programs. The idea of reusing information learned during analysis of the previous program version was employed in [12], [13], [14]. The approaches in [13], [14] store the entire abstract reachable state space and revalidate the affected parts after a change. Our approach works on a function level and stores only the summaries and not the entire abstract state space.

In [12], the authors study substitutability of updated components of a system. Their algorithm is based on inclusion of behaviors and uses a CEGAR loop combining over- and under-approximations of the component behaviors. First, a containment check is performed, ensuring behaviors of the old component occur also in the new one. Second, learning-based assume-guarantee reasoning algorithm is used to to check compatibility, i.e., that the new component satisfies a given property when the old component does. When compared, our approach focuses on real low-level properties of code expressed as assertions rather than abstract inclusion of behaviors. The use of interpolants also appears to be a more practical approach as compared to the application of learning regular languages techniques employed in [12].

The authors of [15] study effects of code changes on function summaries used in compositional directed testing (white-box fuzzing), using must summaries as an under-approximation of the behavior. Their goal is to identify summaries that are affected by the change and cannot be used to analyze the new version. The actual testing is performed using the preserved summaries. In contrast, our algorithm uses over-approximative interpolation-based function summaries and performs the actual verification during the analysis.

Another group of related work aims at equivalence checking of programs [16], [17], [18]. Differential Symbolic Execution [16] attempts to show equivalence of two versions of code using symbolic execution or to compute a behavioral delta when not equivalent. The comparison is function-by-function,

Table II: Experimental evaluation

benchmark		bootstrap		1st change					2nd change				
name	NoI	Total [s]	Itp [s]	IC	Total [s]	Diff [s]	Itp [s]	**Speedup**	IC	Total [s]	Diff [s]	Itp [s]	**Speedup**
`VTT_A`	329	4.889	0.133	2	0.318	0.006	<0.001	**15.6**x	10	15.102	0.006	—	**0.3**x
`VTT_B`	332	23.178	0.003	6	7.793	0.007	0.007	**3.0**x	6	7.805	0.007	0.014	**3.0**x
`VTT_C`	129	0.144	0.001	2	0.017	0.002	<0.001	**8.4**x	1	0.187	0.002	—	**0.8**x
`VTT_D`	247	24.735	0.001	0	0.008	0.008	<0.001	**3098.0**x	2	46.910	0.006	—	**0.8**x
`floppy_A`	292	1.025	0.015	2	0.039	0.009	0.002	**26.1**x	6	0.201	0.009	0.013	**5.0**x
`floppy_B`	294	0.763	0.003	2	0.038	0.009	<0.001	**19.8**x	7	0.046	0.009	0.001	**16.4**x
`floppy_C`	2082	1.280	0.004	2	0.383	0.182	<0.001	**3.4**x	7	0.394	0.183	0.001	**3.2**x
`floppy_D`	2099	60.469	0.257	6	0.374	0.182	<0.001	**161.7**x	23	3.614	0.189	—	**16.8**x
`kbfiltr_A`	529	1.307	0.014	2	0.030	0.011	<0.001	**43.1**x	6	0.111	0.012	0.006	**10.6**x
`kbfiltr_B`	529	1.040	0.001	1	0.052	0.011	0.001	**19.6**x	2	1.835	0.011	—	**0.6**x
`kbfiltr_C`	1010	2.522	0.014	2	0.063	0.021	0.002	**40.2**x	23	0.124	0.021	0.002	**20.3**x
`kbfiltr_D`	1011	3.060	0.009	2	0.061	0.022	<0.001	**50.5**x	7	0.231	0.022	0.003	**7.0**x
`diskperf_A`	486	1.028	0.001	1	0.033	0.008	<0.001	**31.3**x	2	1.751	0.008	—	**0.6**x
`diskperf_B`	492	2.580	0.049	2	0.091	0.009	0.006	**28.3**x	12	2.468	0.009	0.029	**1.1**x
`diskperf_C`	1664	1.126	0.001	1	0.072	0.034	<0.001	**15.6**x	4	0.097	0.034	0.001	**11.5**x
`diskperf_D`	1685	38.609	1.179	1	0.295	0.035	0.016	**130.4**x	2	0.508	0.035	0.020	**75.7**x

the unchanged portions of code are abstracted by the same uninterpreted functions. A similar approach is implemented in the SymDiff tool [17], which decides conditional partial equivalence, i.e., equivalence under certain input constraints. Moreover, SymDiff also allows extraction of the constraints and reports them to the user. Regression Verification [18] employs model checking to prove partial equivalence of programs. As in our algorithm, regression verification starts with syntactic difference check, that identifies the set of modified functions. Then it also traverses the call graph bottom-up, and separately checks equivalence between the old and new versions of the functions, while other functions are abstracted again using the same uninterpreted functions. In these approaches, if the versions do differ, the user is alerted and possibly informed what the different output is and for which input it occurs. For evolving systems, the versions almost always differ and thus the user is distracted by many such reports. In contrast, our algorithm focuses on checking safety of the versions with respect to assertion violation and the user is only alerted when a new violation is introduced by the change. Also, our approach may skip processing parts of the program, if they do not influence safety of the code.

Last group of related work includes approaches using interpolation-based function summaries (such as [4], [5], [6]). Although these do not consider upgrade checking, we believe that our incremental algorithm may be instantiated in their context similar to how we instantiated it in the context of [7].

VI. Conclusion

We presented a new upgrade checking algorithm using interpolation-based function summaries. Instead of model checking the entire new version of a program, the modified functions are first compared against their over-approximative summaries from the old version. If this local check succeeds, the upgrade is safe. We proved that the proposed algorithm is sound, if the summaries are generated from the same proof using the original Pudlák's algorithm. Experimental evaluation using our prototype implementation supports our intuition about ability to check system upgrades locally and demonstrates that the algorithm significantly speeds up checking programs with incremental changes.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS '99*, vol. 1579 of *LNCS*, pp. 193–207, 1999.

[2] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in *Computer Aided Verification, CAV '97*, LNCS, pp. 72–83, 1997.

[3] K. L. McMillan, "Applications of Craig Interpolation in Model Checking," in *TACAS '05*, vol. 3440 of *LNCS*, pp. 1–12, 2005.

[4] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV '06*, vol. 4144 of *LNCS*, pp. 123–136, 2006.

[5] K. L. McMillan, "Lazy annotation for program testing and verification," in *CAV' 10*, vol. 6174 of *LNCS*, pp. 104–118, 2010.

[6] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Whale: An Interpolation-Based Algorithm for Inter-procedural Verification," in *VMCAI '12*, vol. 7148 of *LNCS*, pp. 39–55, 2012.

[7] O. Sery, G. Fedyukovich, and N. Sharygina, "Interpolation-based function summaries in bounded model checking," in *HVC '11*, vol. 7261 of *LNCS*, 2012.

[8] D. Babić and A. J. Hu, "Structural Abstraction of Software Verification Conditions," in *CAV '07*, vol. 4590 of *LNCS*, pp. 371–383, 2007.

[9] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *J. of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.

[10] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.

[11] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *TACAS '10*, vol. 6015 of *LNCS*, pp. 150–153, 2010.

[12] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha, "Dynamic Component Substitutability Analysis," in *FM '05*, vol. 3582 of *LNCS*, pp. 512–528, Springer, 2005.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, "Extreme Model Checking," in *Verification: Theory and Practice*, vol. 2772 of *LNCS*, pp. 332–358, 2003.

[14] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *CAV '05*, vol. 3576 of *LNCS*, pp. 449–461, 2005.

[15] P. Godefroid, S. K. Lahiri, and C. Rubio-González, "Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation," in *SAS '11*, vol. 6887 of *LNCS*, 2011.

[16] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *FSE '08*, pp. 226–237, 2008.

[17] M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Conditional equivalence," Tech. Rep. MSR-TR-2010-119, Microsoft Research, 2010.

[18] B. Godlin and O. Strichman, "Regression verification," in *DAC '09*, pp. 466–471, 2009.

Verification of Parametric System Designs

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri
Fondazione Bruno Kessler, Italy.
Via Sommarive 18, I-38050, Trento, Italy
{cimatti,narasamdya,roveri}@fbk.eu

***Abstract*—System designs are often modeled as sets of threads whose activations are controlled by a domain-specific scheduler. Especially in the early design phases, the interactions between the threads and the scheduler often depend on parameters (such as the duration of thread suspensions) for which a value is not available.**

In this paper, we tackle the verification of designs with parametric scheduler-thread interaction. We propose a new method, called Semi-Symbolic Scheduler/Symbolic Threads (S3ST), to prove that a design satisfies the specified assertions for all possible values of the interaction parameters. We build on Explicit-Scheduler/Symbolic-Threads (ESST), an effective technique for verifying designs with cooperative scheduling, that is however limited to the case of non-parametric interactions. As in ESST, S3ST analyzes each thread symbolically using lazy predicate abstraction. The key difference is in the way the scheduler is dealt with. In ESST, the scheduler is directly executed, using techniques similar to explicit-state model checking. In S3ST, the scheduler is analyzed by combining concrete execution of parts of its state, with the evolution of a symbolically represented set of configurations of interaction parameters.

We have implemented S3ST in the KRATOS software model checker, and have performed an experimental evaluation on a significant set of benchmarks with parametric scheduler-thread interaction. The results clearly demonstrate the effectiveness of the new approach.

I. Introduction

System designs, in many embedded-system settings, are becoming software. Such designs are amenable for high-speed simulations before synthesizing the hardware description. The software typically consists of a set of threads that are activated by a scheduler that implements a set of domain-specific rules. Particularly relevant are multi-threaded software with *cooperative* (or *non-preemptive*) scheduling policy: a thread executes, without any interruption, until it either terminates or explicitly yields the control to the scheduler.

Especially in the early stages of the development process, system designs feature parametric interactions between threads and scheduler. For example, when a thread suspends itself, it does so by calling a suitable scheduler primitive, specifying the duration of the suspension as argument. However, such duration is not necessarily a known numerical constant, but may be modeled in the design by a parameter for which the designer has not selected a value. Thus, the verification process must show that the required properties hold for all subsequent choices of values to the interaction parameters. We also remark that the semantics of some important high-level design languages (e.g. SystemC [1]) allows the parameters of the scheduler-thread interaction primitives to range over reals, thus resulting in timed traces. Restricting the parameters to range over the integers (representing numbers of cycles of fixed duration) is only an approximation.

The problem of model checking cooperative threads has been recently tackled with *Explicit-Scheduler/Symbolic-Threads* (ESST) [2]). ESST combines explicit state techniques to analyze the scheduler with symbolic techniques, based on the lazy predicate abstraction [3], to analyze the threads. ESST orchestrates the analysis of the threads by the direct execution of the scheduler. The threads communicate with each other through shared variables, and communicate/interact with the scheduler (e.g., querying and updating scheduler states) by calling primitive functions provided by ESST. ESST is not able to verify designs with parametric scheduler-thread interactions. In fact, the ability to directly execute the scheduler during the search follows directly from the assumption that the values for the interaction parameters are statically determined.

In this paper, we overcome the limitation of ESST by proposing a new technique, called *Semi-Symbolic Scheduler/Symbolic Threads* (S3ST), that is able to deal with parametric thread-scheduler interactions.

Similar to ESST, in S3ST the threads are analyzed by means of the lazy predicate abstraction. The key difference is that the scheduler, instead of being explicitly executed, is dealt with in a semi-symbolic manner, by combining concrete execution of parts of its state, with the evolution of a symbolically represented set of configurations of interaction parameters.

The approach is based on the following steps. First, we introduce a symbolic representation of time delays for each event, and further abstract the time delays of event notifications with the relations between the symbolic representations. Such an abstraction is carried out by the symbolic analysis, and is passed to the scheduler when it is run. Second, we enable the scheduler to perform reasoning on the relations between the symbolic representations of the time delays. This reasoning determines which event notifications should be triggered at the earliest future time. This can be reduced to checking the satisfiability modulo theory (SMT) [4] of formulas that symbolically represent sets of possible time delays. Third, we enable symbolic analysis, via the lazy predicate abstraction, on the part of the scheduler that modifies the time delays. The part concerns the phase of the scheduler that accelerates the simulation time. This step is non-trivial, because the scheduler must operate on both concrete and symbolic data.

The introduction of ESST was originally motivated by the attempt to avoid performing the lazy predicate abstraction on the scheduler. In order not to lose the ESST advantages, it

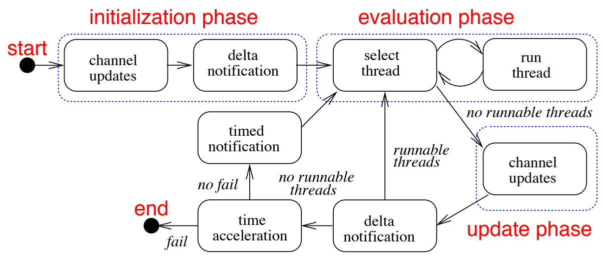


Fig. 1. The SystemC scheduler.

is necessary to control the interactions between the concrete and symbolic data during the scheduler runs. We introduce a technique for predicate filtering, that carefully determines which predicates are relevant to compute the evolution of the scheduler.

In the following we focus on parametric SystemC designs, whose parameters can determine the amount of time delays of event notifications. We have implemented the S3ST algorithm within the KRATOS software model checker [5]. We performed an experimental evaluation on a significant set of new benchmarks and benchmarks adapted from [6], [7] that stress S3ST algorithm. In the experimental evaluation we compare S3ST against the sequentialization approach [2], where the verification problem is reduced to the problem of verifying a sequential program. We also compare S3ST against ESST, by generating non-parametric threaded designs by random sampling the space of parameters. The results of experiments show the effectiveness of S3ST, not only for verification, but also for bug finding.

The paper is organized as follows. Section II provides a background on SystemC and overviews the ESST algorithm. Section III explain the inability of ESST to handle designs with parametric event-notification time delays. Section IV describes the proposed extension to the ESST algorithm. Section V describes some related work. Section VI presents the results of the experimental evaluation. Finally, Section VII concludes this paper and outlines some future work.

II. Background

SystemC is a C++ library that consists of a core language for modeling the components of a system design and their interconnections, and a simulation kernel (or a scheduler) for fast simulations of the design. The core language models system components by means of modules (or C++ classes) and abstracts communication between modules by means of channels. SystemC provides several primitive channels such as signal, mutex, semaphore, and queue. A module can have one or more thread definitions that model the parallel behavior of the system design. The core language provides general-purpose events as synchronization mechanisms between threads.

The SystemC scheduler runs the threads during simulations. Following the SystemC semantics in [1], the scheduler consists of several phases (see Figure 1). In the *initialization phase* all channels are initialized. The scheduler then enters the *evaluation phase* where it executes all runnable threads while postponing the materialization of channel updates performed by the threads. This phase employs a cooperative scheduling policy with mutually-exclusive thread execution. When there are no more runnable threads, the scheduler goes into the *update phase* where it materializes all channel updates postponed during the evaluation phase. An evaluation phase followed by an update phase constitutes a *delta cycle*. A thread, during its execution, can perform delayed event notifications. That is, the involved events will be notified at some time in the future, including at the *delta notification*. The materializations of channel updates also often require the events associated with the updated channels to be notified at the delta notifications. In turn, all threads that are waiting for the notified events or are sensitive to the channels whose associated events are notified become runnable. If, after the delta notification, there are runnable threads, the scheduler goes back to the evaluation phase to run them. Otherwise, it accelerates the simulation time to the nearest time point where there exist events to be notified. These events are then notified at the *timed notification*. Similar to the delta notification, some waiting threads can become runnable after the timed notifications, and thus the scheduler has to go back to the evaluation phase to run them. If there are no more events to be notified at some future time, denoted in Figure 1 by failure in time acceleration, then the simulation ends.

SystemC provides several synchronization functions. For example, when a thread calls `wait(`e`)` for an event e, then the thread suspends itself and waits for the notification of e. If another thread calls e`.notify()`, then all threads waiting for the notification of e are made runnable immediately during the current delta cycle. Event notifications can be delayed. If a thread calls e`.notify(`t`)`, for a time t, then e will not be notified immediately. If t is a constant zero, then e will be notified at the delta-notification, otherwise it will be notified after the simulation time accelerates t time units. Similarly, if a thread calls `wait(`t`)`, then it suspends itself and will become runnable at the timed notification after the simulation time accelerates t time units.

Explicit-Scheduler/Symbolic-Threads (ESST) [2], [8] is an effective technique for the verification of *shared-variable* multi-threaded software with *cooperative* scheduling and *mutually-exclusive* thread executions. The threads communicate with each other through shared variables, and communicate with the scheduler (e.g., querying and updating scheduler states) through a set of *primitive functions* provided by ESST.

ESST is a counter-example guided abstraction refinement (CEGAR) [9] technique that combines explicit and symbolic model checking techniques. It analyzes each thread with the lazy predicate abstraction [3], and orchestrates the whole verification by the direct execution of the scheduler using techniques similar to explicit-state model checking. That is, ESST keeps track of the state of the scheduler explicitly, and includes the scheduler as part of the verification algorithm. For the direct execution of the scheduler, ESST needs precise scheduler states, and thus it requires the arguments passed to the primitive function calls to be constants. Both the scheduler and the set of primitive functions are left abstract, but they are required to exhibit a cooperative scheduling policy.

The ESST algorithm is based on the construction and analysis of an *abstract reachability forest* (ARF) that describes the reachable abstract states of the multi-threaded program. An ARF consists of connected abstract reachability trees (ART's), each of which is obtained by unwinding the control-flow graph (CFG) of the running thread. For a program with N threads, an ARF *node* is a tuple $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$, where l_i and φ_i are, respectively, the location and the region of thread i, φ is the global region, and $\mathbb{S}$ is the scheduler state. Regions are formulas describing the values of program variables, while the scheduler state maintains information about the states of the threads as a mapping from scheduler variables to concrete values.

An ARF is constructed by unwinding the CFGs of threads, and by executing the scheduler. Each ART in the ARF is constructed using the lazy predicate abstraction as for the case of sequential programs. In particular, when the operation of the unwound CFG edge involves a call to a primitive function, then ESST has a *primitive executor* that takes as inputs the scheduler state and the call to a primitive function, and returns the updated scheduler state obtained from directly executing the function call.

Given a node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_i, \varphi_i\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$, such that there are no running threads indicated by $\mathbb{S}$, ESST runs the scheduler on $\mathbb{S}$. The scheduler itself is a function that takes a scheduler state $\mathbb{S}$ (with no running thread) as an input and outputs a set $\{\mathbb{S}'_1, \ldots, \mathbb{S}'_m\}$ of scheduler states representing all possible schedules such that there is only one running thread in $\mathbb{S}'_i$ for $i = 0, \ldots, m$. Each of these states forms an ARF node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_i, \varphi_i\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S}'_{\mathrm{j}})$, that becomes the root of a new ART of the subsequent running thread. Coverage checks and refinements in ESST are similar to that of the lazy predicate abstraction. In particular, the subsumption checks are done thread-wise and require the scheduler states to coincide. We refer the reader to [2], [8] for the details on coverage checks and on the ARF refinement techniques.

To verify SystemC designs, we specialize ESST to SystemC by instantiating the ESST scheduler with the SystemC scheduler, and by defining a set of primitive functions that implement the synchronization functions of SystemC. For example, for an event e and a time t, the SystemC synchronization functions `wait(`e`)`, `wait(`t`)`, and e`.notify(`t`)` correspond, respectively, to the primitive functions `wait_event(`e`)`, `wait_time(`t`)`, and `notify_event(`e`,`t`)`.

III. Parametric Thread-Scheduler Interactions

We focus on parametric designs where the values for the parameters can control the interaction between the threads and the scheduler. In particular we are interested in verifying parametric SystemC designs where the values of parameters determine the amount of delays of event notifications. For example, the design can contain a call `notify_event(`e`,`t`)` or `wait_time(`t`)` where t is non-constant and its values depend on the value of some parameters. Subsequently, we refer to such a form of design as *SystemC designs with parametric event-notification time delays*.

To verify SystemC designs, ESST maintains information about threads and events in the scheduler state. For each thread T, the domain of the scheduler state includes the scheduler variables st_T and ev_T that keep track of, respectively, the state of T and the event whose notification is awaited by T. The variable st_T ranges over the enumerations $\{\mathit{Waiting}, \mathit{Runnable}, \mathit{Running}\}$, whose meanings are obvious. The variable ev_T ranges over the events in the design and are relevant only when st_T is $\mathit{Waiting}$. For each thread T, we implicitly introduce an event e_T whose notification is awaited by the thread when it suspends itself, e.g., by calling the timed wait function `wait_time(`t`)`, for a time t.

For each event e, the domain of scheduler states includes the scheduler variables st_e and $time_e$ that keep track of, respectively, the state and the notification time delay of e. The variable st_e ranges over the enumerations $\{\mathit{Notified}, \mathit{Delta}, \mathit{Timed}, \mathit{None}\}$, where $\mathit{Notified}$ indicates that the event is notified, Delta and Timed indicate that the event will be notified at, respectively, the delta notification and the timed notification, and None indicates that there is no notification. The value of $time_e$ is a concrete time that ranges over $\mathbb{R}^{\geq 0}$ and is relevant only when st_e is Timed.

The parameters that determine the event notification delays may range over $\mathbb{R}^{\geq 0}$. Such parameters cause state explosion. That is, to keep track of such delays, ESST requires infinitely many scheduler states, which in turn needs infinitely many ARF nodes to represent the reachable abstract states. Thus, ESST cannot handle SystemC designs of our interest.

IV. Semi-Symbolic Scheduler/Symbolic Threads

The proposed technique for verifying SystemC designs with parametric event-notification time delays is called S3ST, for *Semi-Symbolic Scheduler/Symbolic Threads*, and is based on the following ideas. First, the threads are analyzed by means of the lazy predicate abstraction technique, in order to build an ARF. Second, the primitive executor is able to handle calls to primitive functions with non-constant time arguments, by enabling the lazy predicate abstraction on the definitions of primitive functions. Third, the scheduler is modeled in such a way that it can perform reasoning on symbolic data carried by the thread and the global regions of ARF nodes. Similar to the primitive executor, the lazy predicate abstraction is enabled on the part of the scheduler that constrains and modifies the time delays, that is, the delta-notification, the timed-notification, and the time acceleration phases. Finally, we ensure that the ARF construction explores all schedules allowed by the possible combinations of event notifications.

Time-Delay Variables. To overcome the state explosion problem described in Section III, we first introduce, for each event e, a *time-delay variable* ϑ_e as a symbolic representation (or a symbolic value) of all possible time delays for the notification of e. The variable $time_e$ in the scheduler state now ranges over $\mathbb{R}^{\geq 0} \cup \{\vartheta_e\}$. For example, in analyzing a call to `wait_time(`t`)` by a thread T, for a non-constant time t, the primitive executor produces an updated scheduler state that maps st_{e_T} to $Timed$, $time_{e_T}$ to ϑ_{e_T}, st_T to $Waiting$, and ev_T to e_T.

Primitive Function Executor. A key idea of our approach is to abstract the time delays of event notifications by the relations between the time-delay variables. These relations are carried by the regions of the ARF nodes, and are analyzed by the lazy predicate abstraction. To this end, we first need to make the time-delay variables visible to the lazy predicate abstraction. Second, we require the primitive executor to provide the lazy predicate abstraction with part of the definition of the called primitive function that updates the time delays.

Let P be a threaded program with N threads, $T_1, \ldots, T_N$. We denote by $SVar$ the set of shared variables of P, by $LVar_T$ the set of local variables of the thread T in P, and by Var_P the set of all variables in P. We assume that $LVar_T \cap SVar = \emptyset$ for every thread T and $LVar_{T_i} \cap LVar_{T_j} = \emptyset$ for each two different threads T_i and T_j. To make time-delay variables visible to the lazy predicate abstraction, we consider them as being shared variables in P. That is, given a set $\{e_0, \ldots, e_m\}$ of events in P, we have $\{\vartheta_{e_0}, \ldots, \vartheta_{e_m}\} \subseteq SVar$. Besides an updated scheduler state, the primitive executor generates on-the-fly a loop-free program defining the update of the time-delay variable. This program is then analyzed by the lazy predicate abstraction.

Let $SState$ be the set of scheduler states, $PrimCall$ be the set of primitive function calls, and $LFProg_P$ be the set of loop-free programs over the variables in Var_P. For simplicity of presentation, we assume that primitive functions do not return any value. The primitive executor for P in S3ST is the function

$$\textsc{Sexec} : (SState \times PrimCall) \rightarrow (SState \times LFProg_P)$$

that takes a scheduler state and a primitive function call as input, and outputs an updated scheduler state along with a loop-free program. For example, in executing `wait_time(`exp`)` called by a thread T, for some expression exp, besides outputting an updated scheduler state, as explained before, the primitive executor generates the program

$$\texttt{assume(}exp >= 0\texttt{);}\ \vartheta_{e_T}\ \texttt{:=}\ exp.$$

Note that, the time-delay variables are viewed as symbolic values by scheduler states, but as program variables by the lazy predicate abstraction.

The scheduler to determine which events to notify at the delta- and timed-notification needs to know the relations among the time-delay variables of the events with constant and non-constant delays. Thus, to enable lazy predicate abstraction to discover predicates that speak about such relations, even if the time delays are constants, the primitive executor always generates the loop-free program.

Semi-Symbolic Scheduler. The scheduler consists of the phases shown in Figure 1. Particularly, in the delta- and timed-notification the scheduler has to reason about the relations between time-delay variables to determine which events to notify. Due to the parameters that affect the time delays, there can be more than one combination of events that can be notified in those phases. Different combinations can result in the simulation time being accelerated to different earliest future times. The scheduler though must allow for the exploration of all possible combinations. Moreover, similar to the primitive executor, because the time acceleration essentially updates the time delays, the scheduler must generate on-the-fly programs representing the updates of the time-delay variables.

The S3ST scheduler is the function

$$\textsc{Sched} : ARFNode \rightarrow \mathcal{P}(\mathcal{P}(SState) \times LFProg_P)$$

that takes an ARF node η as an input and returns a set $\{(S_1, P_1^{lf}), \ldots, (S_n, P_n^{lf})\}$ where S_i is a set of scheduler states and P_i^{lf} is a loop-free program. Particularly for SystemC verification, each S_i is a result of notifying a different set of events in the delta- or timed-notification. In what follows we focus on the timed-notification of the scheduler; the delta-notification can be explained in a similar way.

We denote by $\mathbb{S}[x_0 \mapsto v_0, \ldots, x_n \mapsto v_n]$ the scheduler state obtained from a scheduler state $\mathbb{S}$ by replacing the images of x_i in $\mathbb{S}$ with v_i for $i = 0, \ldots, n$. For simplicity of presentation, we assume that the relations over the time-delay variables and over the parameters are tracked by the global regions of the ARF nodes.

The procedure TIMEDNOTIFICATION shown in Algorithm 1 implements the time acceleration and the timed-notification of Figure 1. Let $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ be the input ARF node, and let TE be the set of events with $Timed$ state. TIMEDNOTIFICATION checks for every non-empty subset $E = \{e_0, \ldots, e_m\}$ of TE whether the events in E can be notified at the same earliest future time, while delaying further the notifications of others in TE. This is done by analyzing the time-delay variables and their relations carried by the global region φ. The analysis amounts to checking, by the procedure SAT, if the conjunction between φ, the equalities $\vartheta_{e_0} = \cdots = \vartheta_{e_m}$, and the inequalities $\vartheta_{e_0} > 0 \wedge \bigwedge\{\vartheta_{e_o} < \vartheta_{e'} \mid e' \in TE \setminus E\}$ is satisfiable. If it is, then the simulation time is accelerated by the procedure ACCELERATETIME, the events in E are notified, and the threads that are waiting for the notifications of the events in E are woken up by the procedure WAKEUPTHREADS, by changing the threads' states from $Waiting$ to $Runnable$.

Intuitively, the procedure TIMEDNOTIFICATION tries all possible combinations of event notifications. If the satisfiability check of the set E is successful, then it means that all events in E can be notified at the same earliest future time, while postponing the notifications of the other events in TE.

The procedure ACCELERATETIME is shown in Algorithm 2. The first for-loop sets the variable $time_e$ of the event e in TE to ϑ_e if $time_{e_0}$ has the symbolic value ϑ_{e_0}. But, note that, if $time_{e_0}$ and $time_e$ are concrete values, then the time acceleration is the same as in the ESST scheduler, i.e., it simply subtracts the value of $time_{e_0}$ from the value of $time_e$ and sets the result as the new value for $time_e$. The pseudo-code following the first for-loop generates a loop-free program P^{lf} that represents the formula checked by SAT, as well

```
Algorithm 1: TIMEDNOTIFICATION
Input  : An ARF node $(\langle l_1,\varphi_1\rangle,\ldots,\langle l_N,\varphi_N\rangle,\varphi,\mathbb{S})$.
Output: A set $R$ of pairs $(\mathbb{S}',P^{lf})$ of a scheduler state $\mathbb{S}'$ and a
        loop-free program $P^{lf}$.
$R \leftarrow \emptyset$
$TE \leftarrow \{e \mid \mathbb{S}(st_e) = Timed\}$
if $TE \neq \emptyset$ then
    for $E \in \mathcal{P}(TE)$ and $E \neq \emptyset$ do
        Let $E = \{e_0,\ldots,e_m\}$
        $Eq \leftarrow \vartheta_{e_0} = \cdots = \vartheta_{e_m}$
        $InEq \leftarrow \vartheta_{e_o} > 0 \wedge \bigwedge\{\vartheta_{e_o} < \vartheta_{e'} \mid e' \in TE \setminus E\}$
        if SAT($\varphi \wedge Eq \wedge InEq$) then
            $(\mathbb{S}',P^{lf}) \leftarrow$ ACCELERATETIME$(\mathbb{S},E)$
            $\mathbb{S}' \leftarrow \mathbb{S}'[st_{e_0} \mapsto Notified,\ldots,st_{e_m} \mapsto Notified]$
            $\mathbb{S}' \leftarrow$ WAKEUPTHREADS$(\mathbb{S}')$
            $\mathbb{S}' \leftarrow \mathbb{S}'[st_{e_0} \mapsto None,\ldots,st_{e_m} \mapsto None]$
            $R \leftarrow R \bigcup \{(\mathbb{S}',P^{lf})\}$
```

```
Algorithm 2: ACCELERATETIME
Input  : A pair $(\mathbb{S},E)$ of a scheduler state $\mathbb{S}$ and a non-empty set $E$ of
        to-be-notified events.
Output: A pair $(\mathbb{S}',P^{lf})$ of a scheduler state $\mathbb{S}'$ and a loop-free
        program $P^{lf}$.
Let $E = \{e_0,\ldots,e_m\}$
$\mathbb{S}' \leftarrow \mathbb{S}$
$TE \leftarrow \{e \mid \mathbb{S}(st_e) = Timed\}$
for $e \in TE$ do
    if $\mathbb{S}(time_{e_0}) = \vartheta_{e_0}$ then $\mathbb{S}' \leftarrow \mathbb{S}'[time_e \mapsto \vartheta_e]$
    else
        if $\mathbb{S}(time_e) \neq \vartheta_e$ then
            $t \leftarrow \mathbb{S}(time_e) - \mathbb{S}(time_{e_0})$
            $\mathbb{S}' \leftarrow \mathbb{S}'[time_e \mapsto t]$
$P^{lf} \leftarrow$ "assume($\vartheta_{e_0} > 0$);"
for $e \in TE$ do
    if $e \in E$ then $P^{lf} \leftarrow P^{lf}$ + "assume($\vartheta_e = \vartheta_{e_0}$);"
    else $P^{lf} \leftarrow P^{lf}$ + "assume($\vartheta_e > \vartheta_{e_0}$);"
    $P^{lf} \leftarrow P^{lf}$ + "$\vartheta_e$ := $\vartheta_e$ - $\vartheta_{e_0}$;"
```

as the updates of time-delay variables caused by the time acceleration. For example, if $E = \{e_0\}$ and $TE = \{e_0, e_1\}$, such that $time_{e_0}$ is mapped to the time-delay variable in the input scheduler state, then the generated loop-free program is:

```
assume($\vartheta_{e_0} > 0$);
assume($\vartheta_{e_0} = \vartheta_{e_0}$); $\vartheta_{e_0}$ := $\vartheta_{e_0}$ - $\vartheta_{e_0}$;
assume($\vartheta_{e_1} > \vartheta_{e_0}$); $\vartheta_{e_1}$ := $\vartheta_{e_1}$ - $\vartheta_{e_0}$;
```

The result of TIMEDNOTIFICATION is a set $\{(\mathbb{S}_1, P_1^{lf}),\ldots,(\mathbb{S}_n, P_n^{lf})\}$ of pairs of a scheduler state and a loop-free program. Each scheduler state $\mathbb{S}_i$ has some runnable threads that must be run in the evaluation phase. That is, for each $\mathbb{S}_i$ such that $st_{T_{i_0}},\ldots,st_{T_{i_m}}$ are mapped to *Runnable*, the scheduler generates a set $S_i = \{\mathbb{S}_i^0,\ldots,\mathbb{S}_i^m\}$ of scheduler states where each $\mathbb{S}_i^j$ is $\mathbb{S}_i[st_{T_{i_j}} \mapsto Running]$. Finally, the scheduler returns the set $\{(S_1, P_1^{lf}),\ldots,(S_n, P_n^{lf})\}$.

ARF Construction. Similar to ESST, the S3ST algorithm is based on the construction of ARF by unwinding the CFGs of threads and by executing the scheduler. Expanding an ARF node involves computing the *abstract strongest post-condition* $SP_{op}^{\pi}(\psi)$ of a region ψ with respect to the operation *op* and the precision π. The operation *op* can be the operation labeling the unwound CFG edge or the loop-free program generated by the primitive executor or by the scheduler. The precision π is a set of predicates that are associated locally with a thread (or a location in the CFG of a thread) or associated with the global region.

We expand an ARF node $\eta = (\langle l_1,\varphi_1\rangle,\ldots,\langle l_N,\varphi_N\rangle,\varphi,\mathbb{S})$ by means of the following rules:

E1. There is a running thread i in $\mathbb{S}$ that performs an operation *op* and (l_i, op, l'_i) is an edge of the CFG of thread i:
- If *op* is *not* a call to a primitive function, then let $\hat{op}$ be *op* and $\mathbb{S}' = \mathbb{S}$.
- If *op* is a call to a primitive function, then $(\mathbb{S}', \hat{op}) =$ SEXEC$(\mathbb{S}, op)$.

The successor node is $(\langle l_1,\varphi'_1\rangle,\ldots,\langle l'_i,\varphi'_i\rangle,\ldots,\langle l_N,\varphi'_N\rangle,\varphi',\mathbb{S}')$, where
- $\varphi'_i = SP_{\hat{op}}^{\pi^{l'_i}}(\varphi_i \wedge \varphi)$,
- $\varphi'_j = SP_{\text{HAVOC}(\hat{op})}^{\pi^{l_j}}(\varphi_j \wedge \varphi)$ for $j \neq i$, and
- $\varphi' = SP_{\hat{op}}^{\pi}(\varphi)$.

The function HAVOC collects all global variables possibly updated by $\hat{op}$, and builds a new operation where these variables are assigned with fresh variables. The precisions π^l and π are associated with the location l of the corresponding CFG and the global region, respectively.

E2. There is *no* running thread in $\mathbb{S}$. For each $(S, P^{lf}) \in$ SCHED(η) and for each scheduler state $\mathbb{S}' \in S$, we create a successor node $(\langle l_1,\varphi'_1\rangle,\ldots,\langle l_N,\varphi'_N\rangle,\varphi',\mathbb{S}')$, where
- $\varphi'_j = SP_{\text{HAVOC}(P^{lf})}^{\pi^{l_j}}(\varphi_j \wedge \varphi)$, for $j = 1,\ldots,n$, and
- $\varphi' = SP_{P^{lf}}^{\pi}(\varphi)$.

such that the successor node becomes the root node of a new ART added to the ARF.

Note that the strongest post-condition with respect to P^{lf} can always be computed because P^{lf} is a loop-free program.

Similar to ESST, the construction of an ARF in S3ST starts with a single ART representing reachable states of the main thread. In the root node of that ART all regions are initialized with *True*, all thread locations are set to the entries of the corresponding threads, and the only running thread in the scheduler state is the main thread.

The ARF is expanded using the rules E1 and E2. An ARF is *complete* if it is closed under the expansion of those rules. An ARF is *safe* if it is complete and, for every node $(\langle l_1,\varphi_1\rangle,\ldots,\langle l_N,\varphi_N\rangle,\varphi,\mathbb{S})$ in the ARF such that $\varphi \wedge \bigwedge_{i=1,\ldots,n}\varphi_i$ is satisfiable, none of the locations $l_1,\ldots,l_N$ are error locations. If one of the locations $l_1,\ldots,l_N$ is an error location, we build a counter-example consisting of paths in the trees of the ARF and check if the counter-example is feasible. Unlike ESST, in the building of counter-example S3ST has to take into account the generated loop-free programs. If the counter-example is feasible, then we have found a real counter-example witnessing that the program is unsafe. Otherwise, we use it to discover predicates to refine the ARF. Coverage checks and refinements of S3ST are the same as those of ESST. We refer to [8] for further details. Note that, because the updates of the time-delay variables are represented by the on-the-fly generated programs that are analyzed symbolically, the existing refinement methods of ESST can discover predicates that speak about the relations between time-delay variables.

Predicate Filtration. One possible bottleneck in S3ST is there can be too many predicates about the relations between time-delay variables that have to be tracked during the ARF construction. The more predicates to track, the more expensive the computations of abstract strongest post-conditions. To alleviate this problem, we perform a predicate filtration that looks up the scheduler state to filter out predicates that contains "inactive" time-delay variables during the computations of abstract strongest post-conditions.

Let q be a predicate and $\mathbb{S}$ be a scheduler state. Denote by $fvar(q)$ the set of free variables occurring in q and by $\Theta(\mathbb{S})$ the set of time-delay variables such that, for each ϑ_e in $\Theta(\mathbb{S})$, we have $\mathbb{S}(st_e) = None$. Given an ARF node $\eta = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ to be expanded with an operation $\hat{op}$ such that the successor scheduler state is $\mathbb{S}'$, then instead of computing the successor global region φ' as $SP^{\pi}_{\hat{op}}(\varphi)$, we compute φ' as $SP^{\pi'}_{\hat{op}}(\varphi)$, where $\pi' = \pi \setminus \{q \in \pi \mid fvar(q) \cap \Theta(\mathbb{S}') \neq \emptyset\}$. The successor thread regions can be computed similarly.

Partial-Order Reduction (POR). POR [10] alleviates the problem of exploring a large number of redundant thread interleavings by exploiting the commutativity of concurrent transitions that result in the same state when they are executed in different orders. The POR techniques developed for ESST in [6] are applicable to S3ST. In [6] we have the procedure PERSISTENT that implements the persistent-set technique. The procedure takes as inputs an ARF node η and a set S of scheduler states resulting from a scheduler run, and outputs a subset of S. For S3ST, we simply run PERSISTENT(η, S_i) for each S_i in $\{(S_1, P^{lf}_1), \ldots, (S_n, P^{lf}_n)\} = $ SCHED(η).

We remark that, the S3ST approach is not a form of POR, particularly because TIMEDNOTIFICATION explores all possible combinations of event notifications. Indeed we can optimize TIMEDNOTIFICATION by techniques inspired by POR. Suppose that we can partition the set of threads in the system design such that in each partition the variables accessed by the threads and the events notified and waited by the threads are disjoint from those of other partitions. Such a partitioning is often possible on a system design that consists of components that do not interact with each other. Given partitions of threads, if a subset E' of E of events to be notified by TIMEDNOTIFICATION wake up threads in partitions different from those woken up by the events in $E \setminus E'$, and the notifications of events in E' can be delayed, then we do not explore the possibility to notify E' together with $E \setminus E'$, but only explore the case where the notification of E' is further delayed.

Correctness. Let S3ST$_{SC}$ be the specialization of S3ST to SystemC, as explained above. In what follows, we assume to work on a threaded program P (representing a SystemC design) with N threads $T_1, \ldots, T_N$. Following the programming framework in [8], a *configuration* γ of P is a tuple $\langle \gamma_{T_1}, \ldots, \gamma_{T_N}, gs, \mathbb{S}\rangle$ where (1) each $\gamma_{T_i} = (l_i, s_i)$ is a thread local configurations, where l_i is a program location and s_i is a mapping (or state) from $LVar_{T_i}$ to values, (2) gs is a mapping (or state) from $SVar$ to values, and (3) $\mathbb{S}$ is a scheduler state. Given a configuration γ and an expression e consisting of variables in $SVar$ and $LVar_{T_i}$ (for $i = 1, \ldots, N$), we denote by $\gamma(e)$ the value resulting from the evaluation of e over γ. The evaluation can be extended naturally to the case of multiple expressions as arguments. For a configuration γ with $\mathbb{S}$ as its scheduler state, we denote by $\gamma[\mathbb{S}'/\mathbb{S}]$ the configuration obtained from γ by replacing $\mathbb{S}$ with a scheduler state $\mathbb{S}'$. Given a state s, we denote by $Dom(s)$ the domain of s. For two states s_1, s_2 with disjoint domains, we denote by $s_1 \cup s_2$ the union s_1 and s_2 such that, for every $x \in Dom(s_1 \cup s_2)$, we have $(s_1 \cup s_2)(x) = s_1(x)$ if $x \in Dom(s_1)$, otherwise $(s_1 \cup s_2)(x) = s_2(x)$. Let φ be a formula over variables in the domain of a state s, we denote by $s \models \varphi$ for a state s satisfying φ.

Let $\eta = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ be an ARF node. We denote by $eq(\mathbb{S})$ the conjunctions of equalities induced by the mappings in $\mathbb{S}$. We say that the configuration $\gamma = \langle (l'_1, s_1), \ldots, (l'_N, s_N), gs, \mathbb{S}'\rangle$ satisfies the node η, denoted by $\gamma \models \eta$, if for all $i = 1, \ldots, N$, we have $l_i = l'_i$, $s_i \cup gs \models \varphi_i$, $\bigcup_{i=1,\ldots,N} s_i \cup gs \models \varphi$, and $\bigcup_{i=1,\ldots,N} s_i \cup gs \cup \mathbb{S}' \models \varphi \wedge eq(\mathbb{S})$.

Following [8], the semantics of each n-ary primitive function f is defined by an $n+1$-ary function $\hat{f}$ that takes as input, in addition to the arguments of f, a scheduler state $\mathbb{S}$ and returns an updated scheduler state $\mathbb{S}'$. For the correctness of S3ST$_{SC}$, we assume that the primitive executor SEXEC implements correctly the definitions of primitive functions. Let η, η' be ARF nodes such that η' is obtained from η by applying rule E1, where the operation op is a call $f(\vec{e})$ to primitive function f with expressions $\vec{e}$ as the arguments. Then, for configurations γ, γ' such that $\gamma \models \eta$, $\mathbb{S}$ is the scheduler state of γ, and $\gamma' = \gamma[\hat{f}(\gamma(\vec{e}), \mathbb{S})/\mathbb{S}]$, we have $\gamma' \models \eta'$.

In what follows, we show that the scheduler SCHED of S3ST$_{SC}$ explores all possible combinations of event notifications. First, let η be an ARF node such that there is no running thread indicated by its scheduler state. Let $(S, P^{lf}) \in$ SCHED(η) and $\mathbb{S} \in S$, we denote by $\eta_{S,\mathbb{S}}$ the successor node obtained from S and $\mathbb{S}$ by rule E2. Second, the SystemC scheduler, as in [8], can be implemented by a function $Sched$ that takes a scheduler state as an input and outputs a set of scheduler states.

Lemma 1: Let η be an ARF node such that there are no running threads in its scheduler state, and let γ be a configuration such that $\gamma \models \eta$ and $\mathbb{S}$ is γ's scheduler state. Let $\hat{S} = Sched(\gamma)$ be the set of scheduler states obtained by running the scheduler. Then, there are a pair $(S, P^{lf}) \in$ SCHED(η) and a one-to-one correspondence C between $\hat{S}$ and S such that, for every scheduler state $\mathbb{S}' \in \hat{S}$, we have $\gamma[\mathbb{S}'/\mathbb{S}] \models \eta_{S,C(\mathbb{S})}$.

Proof: (Sketch) The proof of this lemma relies on the fact that TIMEDNOTIFICATION of SCHED enumerates all possible combinations of event notifications yielded by configurations that satisfy the ARF node η. ■

Intuitively, the above lemma says that, for any configuration that satisfies the ARF node η, the successor configuration obtained by running the scheduler $Sched$ is in the set of configurations represented by the successor abstract state obtained by running $\mathrm{S3ST}_{SC}$'s scheduler SCHED.

The following theorem states the correctness of $\mathrm{S3ST}_{SC}$:

Theorem 1: Let P be a SystemC design with parametric event-notification time delays. For every terminating execution, $\mathrm{S3ST}_{SC}(P)$ returns a safe ARF if and only if P is safe for all possible values of its parameters.

Proof: (Sketch) The proof can be derived from that of ESST in [8]. The correctness of $\mathrm{S3ST}_{SC}$ relies on the above-mentioned assumption about the primitive executor SEXEC and Lemma 1. In particular the computations of abstract strongest post-condition on the on-the-fly generated loop-free programs over-approximate the set of possible values for the time-delay variable ϑ_e of an event e when st_e is $Timed$. ■

V. Related Work

There has been a large amount of work on developing techniques for the verification of both sequential and concurrent (or multi-threaded) programs; see [11] and the related work section of [8] for recent surveys. Most of these techniques do not address timed systems, and assume to deal only with a simple non-deterministic scheduler.

Sequentialization. One popular approach to verifying multi-threaded programs is by means of sequentialization. In this approach the multi-threaded program is translated into a (non-deterministic) sequential program that is behaviorally equivalent, or equivalent up to some bounds (e.g., number of context switches), to the multi-threaded program. The resulting program is then analyzed by off-the-shelf techniques for sequential programs. Our previous work in [2] on sequentializing SystemC designs is already able to handle SystemC designs with parametric event-notification delays because the sequentialization captures the precise semantics of the SystemC scheduler. Indeed, the sequentialization approach can be used to verify general parametric SystemC designs. However, as demonstrated in that paper, the approach does not scale up to large designs.

The work in [12] is concerned with the verification of safety properties of periodic real-time systems with priority-sensitive scheduling. The verification is based on the translation of the system into a sequential program that over-approximates all executions of the system up to some time bound. The resulting sequential program is then verified using bounded model checking (BMC). Similar to our work, the work abstracts time via job-bounded abstraction. However, due to being over-approximations, the analysis of the sequential programs can result in false warnings.

Timed and Hybrid Systems. Other branch of work on the analysis of timed systems is in the context of timed and hybrid systems/automata [13], [14]. The analysis mostly abstracts away data variables, and particularly for timed automata, the analysis cannot handle non-deterministic inputs. Notable exceptions are the SMT-based verification of timed and hybrid automata in [15], [16]. The work in [15] reduces schedulability analysis of parametric timed automata to reachability of error location in the symbolic representation (SMT formulas) of the automata. The reachability analysis is done via BMC and is complete only for periodic systems. The analysis involves neither abstraction nor refinement processes. The work in [16] is concerned with the scenario verification of hybrid systems. Similar to [15], the hybrid systems are represented symbolically as SMT formulas and analyzed by means of BMC.

Path Exploration and Test Case Generation. Techniques that involve mixed symbolic and concrete executions have also been developed in the context of automated path exploration and test cases generations. Popular approaches have been implemented in DART (Directed Automated Random Testing) [17], EXE [18], SPF (Symbolic PathFinder) [19], [20], and S2E [21] DART performs bounded concrete executions on random inputs, while at the same time collects the path constraints of the executed paths. The constraints are then systematically negated to obtain new input values that will direct the next concrete executions to alternative paths. These steps are repeated until the coverage criteria is achieved. EXE and SPF essentially perform symbolic executions, but perform concrete executions to simplify the path constraints. S2E interleaves concrete and symbolic executions. On switching from concrete execution to symbolic one, S2E generalizes the concrete values to symbolic values, and run simultaneously concrete and symbolic executions. On switching in the reverse direction, S2E performs lazy concretization by on-demand instantiations of symbolic data.

VI. Experimental Evaluation

We have implemented the S3ST algorithm, and its specialization to SystemC, in the KRATOS software model checker [5].

Setup. We have carried out an experimental evaluation using new benchmarks and benchmarks derived from [6] and [7]. The derived benchmarks generalize the original ones by adding parameters that control the time delays of event notifications. The number of added parameters corresponds to the number of primitive function calls that concern event notifications (which is linear with the number of threads). For each benchmark x from [6] and [7], we call the derived benchmark p-x. The benchmarks that exhibit thread-scheduler interaction delays that may vary from cycle to cycle are marked with a $\star$.

We compared the S3ST algorithm with the sequentialization approach described in [2]. For the experiments with S3ST, we enabled partial-order reduction. For the sequentialization, we experimented with the lazy predicate abstraction of KRATOS and CPACHECKER SVN revision 6080 [22], the eager abstraction of SATABS-3.0 [23], and the BMC of CBMC-4.0 [24]. For CBMC, we set the number of loop unwinding to 3 and only considered the unsafe benchmarks.

We ran our experiments on an Intel Xeon 3GHz box with 4GB of RAM, and running Linux. We set the time limit to 1000s and the memory limit to 2GB.

Data to reproduce our experiments is available at http://es.fbk.eu/people/roveri/tests/fmcad2012.

Results. Table I shows the results of experiments. The column V shows the status of the benchmarks: S for safe and U for unsafe. For each tool we report the execution time in seconds. We use T.O for out of time, M.O for out of memory, U.R for returning unknown, E.R for having run time errors, and N.A for not available. For S3ST, we performed experiments with and without predicate filtration (resp. columns PF and No-PF).

In general, it is clear that S3ST outperforms the sequentialization techniques. For the sequentialization approach, a close inspection on KRATOS reveals that, even for the small `p-token-ring.2` benchmark, the analysis has to keep track of 45 predicates. For CBMC, the * mark on the results indicate that, due to insufficient loop unwindings, CBMC reports that the benchmarks are safe. Any attempt to increase the number of loop unwindings results in out of time. We also see that the impact of the predicate filtration is very significant for the scalability of S3ST. For example, for `p-token-ring.4` benchmark the predicate filtration, on average, can filter out 44% of predicates used in the abstraction computations. Finally, we notice that the $\star$ benchmarks, featuring cycle-varying parameters, are even harder for sequentialization.

The following table shows the behavior of S3ST when the number of parameters in the benchmarks is increased. We present the results for the `p-token-ring.10` and the `p-toy` benchmarks that have, respectively, 11 and 3 parameters. (Other benchmarks show a similar behavior.)

	p-token-ring.10									p-toy			
#Parameters	0	1	2	3	4	5	6	7	8	0	1	2	3
Run Time	1.6	23.6	27.6	33.4	42.2	61.8	289.5	448.2	743.0	2.5	4.7	99.7	99.8
# ARF Nodes	1378	2513	3422	4673	5941	9324	21110	24558	28018	673	787	4245	4233
#Preds	23	54	56	60	66	74	84	96	110	23	27	55	54

For the `p-token-ring.10` table, the column j shows the experiment on a benchmark obtained from `p-token-ring.10` by concretizing $11 - j$ parameters with some constants. Similarly for the `p-toy` table. The presence of parameters potentially increases the number of thread interleavings that S3ST has to explore, as shown by the number of visited ARF nodes. For the experiments reported in the `p-token-ring.10` table, the predicate filtration is effective in reducing the number of predicates that concern the relations of the time-delay variables: on average, 41.46% reduction. However, S3ST still has to keep track of the predicates that concern the relations between the constraints over the parameters themselves. The more parameters, the more predicates it has to track, as indicated in the row #Pred. Analyzing benchmarks containing both constant and parametric time delays can be as hard as analyzing those containing only parametric time delays. Recall that the scheduler uses the relations between the time-delay variables to determine the events to notify. Thus, even though the time delay of the notification of an event is a constant, S3ST may still have to keep track of predicates containing the time-delay variable associated with that event.

We have also investigated the possibility of analyzing with ESST the benchmarks obtained by grounding the time delay parameters with a number of (random) values. For

TABLE I
RESULTS OF EXPERIMENTAL EVALUATION (IN SEC).

		S3ST		Sequentialization			
Name	V	PF	No-PF	KRATOS	CPA	SATABS	CBMC
p-kundu-bug-1	U	**1.18**	1.19	23.18	U.R	375.04	5.26
p-kundu-bug-2	U	**0.87**	0.89	44.54	U.R	T.O	22.04
p-kundu	S	**54.62**	62.66	T.O	U.R	T.O	N.A
p-mem-slave-tlm.1	S	**10.07**	38.87	T.O	E.R	531.79	N.A
p-mem-slave-tlm.2	S	**54.16**	T.O	T.O	M.O	878.71	N.A
p-mem-slave-tlm.3	S	**185.95**	T.O	T.O	M.O	T.O	N.A
p-mem-slave-tlm.4	S	**517.00**	T.O	T.O	M.O	T.O	N.A
p-mem-slave-tlm.5	-	T.O	T.O	T.O	E.R	T.O	N.A
p-mem-slave-tlm-bug.1	U	**6.65**	25.18	T.O	M.O	T.O	*306.33
p-mem-slave-tlm-bug.2	U	**35.64**	882.55	T.O	M.O	T.O	*286.46
p-mem-slave-tlm-bug.3	U	**106.80**	T.O	T.O	M.O	T.O	*278.67
p-mem-slave-tlm-bug.4	U	**402.51**	T.O	T.O	M.O	T.O	*293.06
p-mem-slave-tlm-bug.5	U	**991.57**	T.O	T.O	M.O	T.O	*323.01
p-mem-slave-tlm-bug2.1	U	**4.23**	4.79	T.O	M.O	T.O	*295.68
p-mem-slave-tlm-bug2.2	U	**15.48**	17.58	T.O	M.O	T.O	*295.17
p-mem-slave-tlm-bug2.3	U	**43.17**	45.87	T.O	M.O	T.O	*283.85
p-mem-slave-tlm-bug2.4	U	**99.73**	104.42	T.O	M.O	T.O	*306.81
p-mem-slave-tlm-bug2.5	U	**236.81**	244.65	T.O	M.O	T.O	*336.84
p-pc-sfifo-1	S	**3.45**	4.29	T.O	U.R	197.29	N.A
p-pc-sfifo-2	S	**3.49**	4.00	239.01	U.R	193.05	N.A
p-token-ring.1	S	**0.56**	0.59	20.97	83.22	904.94	N.A
p-token-ring.2	S	**1.49**	2.09	T.O	M.O	T.O	N.A
p-token-ring.3	S	**3.49**	13.48	T.O	M.O	T.O	N.A
p-token-ring.4	S	**8.08**	430.58	T.O	M.O	T.O	N.A
p-token-ring.5	S	**14.73**	T.O	T.O	M.O	T.O	N.A
p-token-ring.6	S	**27.84**	T.O	T.O	M.O	T.O	N.A
p-token-ring.7	S	**70.53**	T.O	T.O	M.O	T.O	N.A
p-token-ring.8	S	**192.87**	T.O	T.O	E.R	T.O	N.A
p-token-ring.9	S	**789.14**	T.O	T.O	M.O	T.O	N.A
p-token-ring.10	-	T.O	T.O	T.O	M.O	T.O	N.A
p-token-ring-bug.1	U	**0.47**	0.49	14.73	20.59	485.92	6.47
p-token-ring-bug.2	U	**1.18**	1.29	T.O	E.R	773.84	16.62
p-token-ring-bug.3	U	**2.49**	5.09	T.O	M.O	T.O	*33.83
p-token-ring-bug.4	U	**5.68**	95.05	T.O	M.O	T.O	*91.53
p-token-ring-bug.5	U	**9.98**	T.O	T.O	E.R	T.O	*154.57
p-token-ring-bug.6	U	**17.76**	T.O	T.O	M.O	T.O	*250.71
p-token-ring-bug.7	U	**45.55**	T.O	T.O	M.O	T.O	*478.11
p-token-ring-bug.8	U	**106.39**	T.O	T.O	M.O	T.O	*752.50
p-token-ring-bug.9	U	**537.08**	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug.10	U	T.O	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug2.1	U	**0.48**	0.49	14.05	15.51	465.44	6.55
p-token-ring-bug2.2	U	**1.39**	1.59	T.O	M.O	740.50	20.55
p-token-ring-bug2.3	U	**3.56**	6.49	T.O	E.R	T.O	*46.87
p-token-ring-bug2.4	U	**8.76**	103.04	T.O	M.O	T.O	*81.79
p-token-ring-bug2.5	U	**24.66**	T.O	T.O	M.O	T.O	*165.14
p-token-ring-bug2.6	U	**47.55**	T.O	T.O	M.O	T.O	*310.16
p-token-ring-bug2.7	U	**100.49**	T.O	T.O	M.O	T.O	*499.32
p-token-ring-bug2.8	U	**372.45**	T.O	T.O	M.O	T.O	*748.97
p-token-ring-bug2.9	U	**925.20**	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug2.10	-	T.O	T.O	T.O	M.O	T.O	T.O
p-toy-bug-1	U	**13.66**	18.39	T.O	M.O	T.O	*47.09
p-toy-bug-2	U	25.04	**24.88**	T.O	M.O	T.O	*52.23
p-toy	S	**99.90**	T.O	T.O	M.O	T.O	N.A
p-transmitter.1	U	**0.07**	0.09	8.20	7.73	470.19	2.84
p-transmitter.2	U	**0.29**	**0.29**	T.O	E.R	618.49	*8.96
p-transmitter.3	U	**0.49**	**0.49**	T.O	E.R	T.O	*21.49
p-transmitter.4	U	**0.89**	0.99	T.O	M.O	T.O	*43.43
p-transmitter.5	U	**1.59**	1.79	T.O	E.R	T.O	*101.71
p-transmitter.6	U	**2.49**	2.99	T.O	M.O	T.O	*180.11
p-transmitter.7	U	**3.97**	4.89	T.O	M.O	T.O	*299.19
p-transmitter.8	U	**5.89**	7.89	T.O	M.O	T.O	*503.97
p-transmitter.9	U	**8.57**	12.38	T.O	M.O	T.O	*815.18
p-transmitter.10	U	**11.66**	20.08	T.O	M.O	T.O	T.O
rod1-bug.c ⋆	U	**28.89**	34.16	T.O	M.O	T.O	*312.76
rod1.c ⋆	S	**285.18**	308.38	T.O	M.O	T.O	N.A
rod2.c ⋆	S	**76.84**	88.13	T.O	M.O	T.O	N.A
modtrans-bug.c	U	**64.77**	379.59	T.O	M.O	T.O	*643.61
modtrans-nudc-bug1.c ⋆	U	**238.11**	T.O	T.O	M.O	T.O	*583.50
modtrans-nudc-bug2.c ⋆	U	**232.91**	T.O	T.O	M.O	T.O	*569.42
modtrans-nudc-bug3.c ⋆	U	**131.25**	657.79	T.O	M.O	T.O	*584.66
modtrans-nudc-bug4.c ⋆	U	**111.53**	675.21	T.O	M.O	T.O	*604.64
modtrans-nudc1.c ⋆	S	**211.64**	T.O	T.O	M.O	T.O	N.A
modtrans-nudc2.c ⋆	U	**157.75**	649.20	T.O	M.O	T.O	*827.36
modtrans-rec1-bug1.c	U	**4.08**	4.89	T.O	M.O	T.O	*588.06
modtrans-rec1-bug2.c	U	**4.09**	4.90	T.O	M.O	T.O	*602.50
modtrans-rec1.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans-rec2-bug1.c	U	**6.27**	17.58	T.O	M.O	T.O	*585.99
modtrans-rec2-bug2.c	U	**4.49**	6.89	T.O	M.O	T.O	*607.05
modtrans-rec2.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans2-nudc-bug1.c ⋆	U	**143.80**	T.O	T.O	M.O	T.O	*810.07
modtrans2-nudc-bug2.c ⋆	U	**173.04**	T.O	T.O	M.O	T.O	*567.25
modtrans2-nudc-bug3.c ⋆	U	**236.81**	T.O	T.O	M.O	T.O	*564.44
modtrans2-nudc-bug4.c ⋆	U	**237.52**	T.O	T.O	M.O	T.O	*561.79
modtrans2-nudc-bug5.c ⋆	U	**226.81**	T.O	T.O	M.O	T.O	*804.03
modtrans2-nudc1.c ⋆	U	**150.09**	720.50	T.O	M.O	T.O	*557.49
modtrans2-nudc2.c ⋆	U	**159.44**	705.55	T.O	M.O	T.O	*565.91
ss1.c ⋆	U	**4.59**	6.98	T.O	M.O	T.O	*235.45
ss2-bug.c ⋆	U	**1.19**	1.29	38.04	M.O	T.O	*7.12
train-hytech-bug1.c ⋆	U	**0.29**	**0.29**	T.O	M.O	883.89	472.02
train-hytech-bug2.c ⋆	U	**0.29**	0.30	T.O	M.O	995.21	451.60
train-hytech-bug3.c ⋆	U	**13.79**	15.88	T.O	M.O	784.59	492.15
train-hytech-bug4.c ⋆	U	**13.78**	15.68	T.O	M.O	784.22	454.18
train-hytech1.c ⋆	S	**34.46**	57.56	T.O	M.O	787.31	N.A
train-hytech2.c ⋆	S	**34.25**	52.34	T.O	M.O	789.87	N.A
train1-bug.c ⋆	U	**0.99**	1.09	T.O	M.O	T.O	*208.07
train1.c ⋆	S	**1.59**	**1.59**	T.O	M.O	T.O	N.A
train2-bug.c ⋆	U	**2.59**	2.78	T.O	M.O	T.O	*200.73
train2.c ⋆	S	**2.09**	2.19	T.O	M.O	T.O	N.A

TABLE II
RESULTS OF GROUNDING APPROACH VS. S3ST.

	Number of Ground Values (ESST)					S3ST
	2	3	4	5	6	
#Unsafe/Safe/T.O	1/9/0	3/7/0	4/6/0	4/4/2	2/0/8	-
Max. Unsafe Time	12.9	47.2	193.8	700.8	828.7	25.1
Avg. Unsafe Time	12.9	39.9	159.5	457.2	590.6	25.1
Max. Safe Time	12.1	51.8	167.5	388.8	-	-
Avg. Safe Time	5.7	32.4	115.8	305.2	-	-

p-toy-bug-2

example, given a primitive function call `wait_time(`t`)` for a non-constant time t and assume that t ranges over the set $\{v_0, \dots, v_k\}$ of concrete values, we replace the call with the following code:

```
assume(t == v0 || ... || t == vk);
if (t == v0) wait_time(v0); ...;
if (t == vk) wait_time(vk);
```

This is clearly an under-approximation, that can only be used for bug finding. The results for `p-toy-bug-2` are reported on Table II. The column k reports the results of 10 experiments with k concrete values. The table also compares the grounding approach with S3ST (on the original parametric benchmark). The table shows that increasing the number of values may increase the chance to find violations; however, the performance of ESST degrades (and possibly times out), even when it does find the bug.

VII. CONCLUSION AND FUTURE WORK

We have presented a novel approach, called S3ST, to the verification of designs where the interactions between thread and scheduler are parametric. The key feature of the approach is the semi-symbolic analysis of the scheduler, that requires a careful control of the interactions between the concrete and symbolic data. The approach allows us to verify parametric designs that are out of reach for techniques based on sequentialization, and is also competitive for bug finding.

For future work, we want to improve further the scalability of ESST and S3ST by applying symmetry reduction, and to generalize the methods to the case of multi-threaded software that is parameterized on the number of threads.

REFERENCES

[1] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A Temporal Language for SystemC," in *FMCAD*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–9.

[2] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: A software model checking approach," in *FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 51–59.

[3] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Art. Int. and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 825–885.

[5] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos - a software model checker for SystemC," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 310–316.

[6] A. Cimatti, I. Narasamdya, and M. Roveri, "Boosting Lazy Abstraction for SystemC with Partial Order Reduction," in *TACAS*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 341–356.

[7] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri, "An analytic evaluation of SystemC encodings in promela," in *SPIN*, ser. LNCS, A. Groce and M. Musuvathi, Eds., vol. 6823. Springer, 2011, pp. 90–107.

[8] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking with explicit scheduler and symbolic threads," *Journal of Logical Methods in Computer Science*, vol. 8, no. (2:18), 2012, arXiv:1206.3182v2 [cs.LO], http://arxiv.org/abs/1206.3182.

[9] 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.

[10] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. LNCS. Springer, 1996, vol. 1032.

[11] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[12] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-bounded analysis of real-time systems," in *FMCAD*, P. Bjesse and A. Slobodova, Eds. FMCAD Inc, 2011, pp. 72–80.

[13] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE, 2006, pp. 125–126.

[14] R. Alur, "Formal verification of hybrid systems," in *EMSOFT*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds. ACM, 2011, pp. 273–278.

[15] A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic computation of schedulability regions using parametric timed automata," in *IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2008, pp. 80–89.

[16] A. Cimatti, S. Mover, and S. Tonetta, "Efficient scenario verification for hybrid automata," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 317–332.

[17] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[18] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[19] C. S. Pasareanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 34–44.

[20] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 179–180.

[21] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 265–278.

[22] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 184–190.

[23] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *TACAS*, ser. LNCS, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 570–574.

[24] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, ser. LNCS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

Deciding Floating-Point Logic with Systematic Abstraction

Leopold Haller*, Alberto Griggio†, Martin Brain*, Daniel Kroening*
*Computer Science Department, University of Oxford, Oxford, UK
first.last@cs.ox.ac.uk
†Fondazione Bruno Kessler, Trento, Italy
griggio@fbk.eu

***Abstract*—We present a bit-precise decision procedure for the theory of binary floating-point arithmetic. The core of our approach is a non-trivial generalisation of the conflict analysis algorithm used in modern SAT solvers to lattice-based abstractions. Existing complete solvers for floating-point arithmetic employ bit-vector encodings. Propositional solvers based on the Conflict Driven Clause Learning (CDCL) algorithm are then used as a backend. We present a natural-domain SMT approach that lifts the CDCL framework to operate directly over abstractions of floating-point values. We have instantiated our method inside MATHSAT5 with the floating-point interval abstraction. The result is a sound and complete procedure for floating-point arithmetic that outperforms the state-of-the-art significantly on problems that check ranges on numerical variables. Our technique is independent of the specific abstraction and can be applied to problems beyond floating-point satisfiability checking.**

***Index Terms*—floating point, decision procedures, abstract interpretation.**

I. INTRODUCTION

Floating-point computations are used pervasively in low-level control software and embedded applications. Such programs are frequently used in areas where safety is of critical importance, such as the automotive and aerospace industry.

Floating-point numbers have a dual nature that complicates complete logical reasoning. On the one hand, they are approximate representations of real numbers, suggesting a numeric approach to their analysis. On the other hand, their discrete nature leads to "odd behaviours", which purely numeric techniques are ill-equipped to handle.

Current complete satisfiability decision procedures for constraints over floating-point numbers are based on bit-vector encodings [1]. The resulting instances are often hard for current Satisfiability Modulo Theory (SMT) solvers. On the other hand, inexpensive techniques such as floating-point interval propagation [2] can be employed to solve *some* instances very efficiently.

To illustrate this point, consider the formula $x \in [0.0, 10.0] \wedge y = x^5 \wedge y > 10^5$, over double-precision floating-point variables x and y. Interval propagation can deduce in a fraction of a second that $y \in [0.0, 100000.0]$ holds, which contradicts the final conjunct $y > 10^5$. In stark contrast, the SMT solver Z3 requires 16 minutes on a modern processor to prove unsatisfiability of a corresponding bit-vector encoding. Likewise, it is possible to construct very simple formulas that interval propagation cannot solve: Consider the floating-point formula $z = y \wedge x = y \cdot z \wedge x < 0$. Standard interval propagation cannot determine that $y \cdot z$ must be positive and fails to prove unsatisfiability. Z3 solves the problem above in less than a second.

The power of an incomplete proof technique such as interval propagation can be boosted by decomposing the proof attempt into cases. In classic DPLL(T) [3], for example, a SAT solver based on the Conflict Driven Clause Learning (CDCL) algorithm enumerates cases by assigning predicates occurring in the formula to candidate truth values. A separate theory decision procedure is then used to check whether the resulting cases are consistent.

In the above examples, classic DPLL(T) would not be able to provide a further refinement since all predicates must be true for the formulas to be satisfiable. However, further decomposition into cases is still possible if we directly enter the domain of the theory. If we assume that $y < 0$ it follows that $z < 0$, which is sufficient to show that $x > 0$. The complementary case $y \geq 0$ can be shown with similar ease. A complete procedure can be obtained in this way since i) interval propagation is complete for sufficiently small cases, e.g., the case where every variable is assigned to a singleton range and ii) there is a finite number of such cases that need to be checked.

In essence, it is possible to use the DPLL(T) framework to perform case splitting directly in the theory [4]. This requires introduction of a potentially large number of new propositions to represent theory facts and makes implementation of good learning heuristics difficult, since the propositional learning algorithm is unaware of the theory semantics associated with propositions. To handle problems such as the above, the emerging area of *natural domain* SMT procedures [5]–[9] aims at increasing the power of SMT techniques by lifting them directly to richer logics. For example, where a CDCL solver makes *decisions* that force Boolean variable to true or false, a natural domain SMT solver for linear integer arithmetic may set a variable to some specific integer value [6]. Such procedures

Supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, and the FP7 STREP PINCETTE.
† 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.

typically require custom, domain-specific decision heuristics and learning procedures.

The work presented in this paper can be seen as a systematic derivation of a learning algorithm for floating-point logic from an abstract domain. We exploit a simple insight, advocated in an earlier paper [10]: Propositional SAT solvers internally operate over a lattice-theoretic abstraction that overapproximates the space of possible solutions. Natural liftings of CDCL-style learning to richer logics can be obtained by considering a wider scope of abstractions.

In this paper, we show how the FIRST-UIP learning algorithm [11] used in CDCL solvers can be lifted to a wider range of domains. This lifting is non-trivial since it has to address the additional complexity of abstractions for domains that go beyond propositional logic. We present a new implementation of our approach for floating-point logic as part of the MATHSAT5 framework. The implementation outperforms approaches based on bit-blasting significantly on our set of benchmarks.

Contribution: The contributions of this paper are threefold: *(i)* we present a novel natural domain solver for the theory of floating-point arithmetic that significantly outperforms the state of the art; *(ii)* we introduce a lifting of the FIRST-UIP conflict analysis algorithm used in modern SAT solvers to abstractions, *(iii)* we evaluate our work on a set of benchmarks.

Outline: Section II provides a brief introduction to floating-point numbers, the theory of floating-point arithmetic and some formal background on abstract interpretation. Section III gives a high-level account of model search and conflict analysis over abstract domains. The main algorithmic contribution is presented in Section IV: A lifting of the FIRST-UIP algorithm to abstract domains. The implementation of our floating-point solver, the specific heuristics we used and experiments are discussed in Section V. An extensive survey of related work from the areas of theorem proving, abstract interpretation, and decision procedures is given in Section VI.

II. Floating-Point Arithmetic and Abstraction

A. Floating-Point Arithmetic

This section gives an informal introduction to the theory of floating-point arithmetic. For an exhaustive treatment, see [12] which formalises the IEEE-754 floating-point standard as an SMT theory.

Floating-point numbers are approximate representations of the reals that allow for fixed size bit-vector encoding. A floating-point number represents a real number as a triple of positive integers (s, m, e), consisting of a *sign bit* s taken from the set of Booleans $\mathbb{B} \hat{=} \{0, 1\}$, a *significand* m and an *exponent* e. Its real interpretation is given by $(-1)^s \cdot m \cdot 2^e$. Note that all numbers have a sign, therefore, the real number 0 is represented both by an *unsigned zero* $+0$ and a *signed zero* -0.

A floating-point format determines the number of bits used for encoding significand and exponent. The IEEE-754 standard defines several floating-point formats and their bit-encodings. An example of an IEEE-754 `binary16` floating-point number is given below.

$$\underbrace{\boxed{1}}_{s}\underbrace{\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}}_{e}\underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}}_{m}\begin{array}{l} = -1 \cdot 2^{18-15} \cdot 1.3359375 \\ = -10.6875 \end{array}$$

Some bit-patterns are used to encode the *special values* positive infinity $+\infty$, negative infinity $-\infty$, and NaN, which represents an invalid arithmetic result. We do not go into details regarding this encoding and simply define $\mathbb{F}$ to be the set of all floating-point numbers including the special values.

Terms in FPA are constructed from floating-point variables, constants, standard arithmetic operators and special operators such as square roots and combined multiply-accumulate operations used in signal processing. Most operations are parameterized by one of five rounding modes. The result of floating-point operations is defined to be the real result (computed with 'infinite precision') rounded to a floating-point number using the chosen rounding mode.

Formulas in FPA are Boolean combinations of predicates over floating-point terms. In addition to the standard equality predicate $=$, FPA offers a number of floating-point specific predicates including a special floating-point equality $=_{\mathbb{F}}$, and floating-point specific arithmetic inequalities $<$ and $\leq$. Since these operators approximate real comparisons they have unusual properties. For example, any comparison with the value NaN returns false, therefore $=_{\mathbb{F}}$ is not reflexive since $NaN =_{\mathbb{F}} NaN$ does not hold. On the other hand, $+0$ and -0 compare as equal since they represent the same real number.

B. Lattices and Abstractions

Following the theory of abstract interpretation [13] we define abstraction in terms of lattices and closure operators. A *complete lattice* is a partially ordered set $(P, \sqsubseteq)$ in which any subset $S \subseteq P$ has a unique least upper bound $\bigsqcup S$ and unique greatest lower bound $\bigsqcap S$. A complete lattice has a least element $\bot$ and a greatest element $\top$. A *powerset lattice* of a set Q is a complete lattice $(\wp(Q), \subseteq)$ with least upper bound $\bigcup$, and greatest lower bound $\bigcap$. A *transformer* on P is a monotone function $f : P \to P$. Transformers over P form a complete lattice under the pointwise order, $f \sqsubseteq g$ if $\forall p \in P. f(p) \sqsubseteq g(p)$. Least upper bounds and greatest lower bounds extend pointwise to the transformer lattice, e.g., $f \sqcap g = \lambda p.\ f(p) \sqcap g(p)$. We denote the least fixed point of a transformer g as $\mathsf{lfp}\, X.\ g(X)$ or $\mathsf{lfp}\, g$, and the greatest fixed point $\mathsf{gfp}\, X.\ g(X)$ or $\mathsf{gfp}\, g$. The *image* of f is the set $Img(f) \hat{=} \{f(p) \mid p \in P\}$.

A *closure operator* on P is a transformer $\zeta : P \to P$ such that for all $p, q \in P$, *(i)* ζ is *extensive*, i.e., $p \sqsubseteq \zeta(p)$ and *(ii)* ζ is *idempotent*, i.e., $\zeta(p) = \zeta(\zeta(p))$. An *abstraction* of a lattice $(P, \sqsubseteq)$ is a complete sublattice $(Q, \sqsubseteq)$ with $Q \subseteq P$, such that $Q = Img(\zeta)$ for some closure operator ζ. We call P the concrete and Q the abstract domain. The closure operator ζ maps a set to its most precise abstract representation. We assume throughout this paper that $\zeta(\bot) = \bot$. An *abstract transformer* $g : Q \to Q$ is an *overapproximation* of a transformer $f : P \to P$ if $\forall q \in Q.\ f(q) \sqsubseteq g(q)$ and an *underapproximation* if $\forall q \in Q.\ g(q) \sqsubseteq f(q)$. The unique *best overapproximation* of $f : P \to P$ w.r.t. to a closure operator ζ is the function $g = \zeta \circ f \circ \zeta$. A best underapproximation does, in general, not exist.

Example II.1 (Intervals for $\wp(\mathbb{F})$)**.** Intervals approximate sets of numbers by their closest enclosing range. In addition to the arithmetic ordering $\leq$, the IEEE-754 standard dictates a total order $\preceq$ over all floating-point values, including special values such as NaN. The interval abstraction is defined by a closure operator $\zeta : \wp(\mathbb{F}) \to \wp(\mathbb{F})$ where $\zeta(S) \mathrel{\hat=} \{v \in \mathbb{F} \mid \min_{\preceq}(S) \preceq v \preceq \max_{\preceq}(S)\}$.

C. Logic and Abstraction

In this section, we summarise our basic framework for model-theoretic approximations of logical formulas using abstraction (see [10] for more details) and show how it applies to FPA. Let *Forms* be the set of *formulas*, *Structs* be a set of semantic *structures*. The semantics of a logic are given as an interpretation function $[\![\cdot]\!]_\cdot : (\mathit{Forms} \times \mathit{Structs}) \to \mathbb{B}$. An element $\sigma \in \mathit{Structs}$ is a *model* of a formula $\varphi \in \mathit{Forms}$ if $[\![\varphi]\!]_\sigma = 1$ and a *countermodel* otherwise. A formula is *satisfiable* if it has a model and *unsatisfiable* otherwise.

Semantic structures in FPA are given by *floating-point assignments*, defined as $\mathit{FloatAsg} \mathrel{\hat=} \mathit{Vars} \to \mathbb{F}$, where *Vars* is a finite set of first-order variables.

For a formula φ, we define two transformers on the powerset lattice $\wp(\mathit{Structs})$.

Definition II.1. The *model transformer* mods_φ and the *conflict transformer* confs_φ are defined as follows.

$$\begin{aligned} \mathit{mods}_\varphi(S) &\mathrel{\hat=} \{\sigma \in \mathit{Structs} \mid \sigma \in S \wedge [\![\varphi]\!]_\sigma = 1\} \\ \mathit{confs}_\varphi(S) &\mathrel{\hat=} \{\sigma \in \mathit{Structs} \mid \sigma \in S \vee [\![\varphi]\!]_\sigma = 0\} \end{aligned}$$

The model transformer maps a set of structures to its smallest subset that contains the same models. The conflict transformer (also referred to as the *universal countermodel transformer* in [10]) maps a set of structures to its largest superset that contains the same models. The model transformer can be used to refine an overapproximation of a set of models, and the conflict transformer to generalise an underapproximate set of countermodels.

Satisfiability can be expressed in terms of these operators. Note that mods_φ and confs_φ are idempotent, therefore $\mathit{mods}_\varphi(\mathit{Structs}) = \mathsf{gfp}\, \mathit{mods}_\varphi$ and $\mathit{confs}_\varphi(\emptyset) = \mathsf{lfp}\, \mathit{confs}_\varphi$.

Theorem 1. *The following statements hold.*
1) $\mathsf{gfp}\, \mathit{mods}_\varphi = \emptyset$ *exactly if* φ *is unsatisfiable.*
2) $\mathsf{lfp}\, \mathit{confs}_\varphi = \mathit{Structs}$ *exactly if* φ *is unsatisfiable.*

We can compute these fixed points abstractly to perform incomplete satisfiability checks. Propositional solvers use the partial assignment abstraction [10]. For example a partial assignment $\langle p{:}\mathsf{true}, q{:}\mathsf{false}\rangle$ abstractly represents the set of assignments σ from propositions to truth values, where $\sigma(p) = \mathsf{true}$ and $\sigma(q) = \mathsf{false}$ and all other propositions may be mapped to either truth value.

In this paper, we use the interval abstraction. Recall that IEEE-754 requires a total ordering $\preceq$. We use it to define an interval abstraction for the powerset lattice $\wp(\mathit{FloatAsg})$. An *interval assignment*, written $\langle x_1 : [l_1, u_1], \ldots, x_k : [l_k, u_k]\rangle$, is a set of floating-point assignments $\{\sigma \mid \forall i.\ l_i \preceq \sigma(x_i) \preceq u_i\}$. We denote the set of all interval assignments by $\mathbb{I}_\mathbb{F}$, which forms a complete lattice under the set order $\subseteq$. The closure operator defining the interval abstraction is given as $\zeta(S) \mathrel{\hat=} \langle x_1 : [l_1, u_1], \ldots, x_k : [l_k, u_k]\rangle$ where $l_i = \min_{\preceq}\{\sigma(x_i) | \sigma \in S\}$ and $u_i = \max_{\preceq}\{\sigma(x_i) | \sigma \in S\}$.

For example, let $f = \{x \mapsto 4.2, y \mapsto 2.3\}$ and $g = \{x \mapsto 1.8, y \mapsto 10.5\}$, then applying ζ yields $\zeta(\{f, g\}) = \langle x : [1.8, 4.2], y : [2.3, 10.5]\rangle$.

We can use the interval abstraction to approximate the fixed points of Theorem 1.

Theorem 2. *Let* amods_φ *be an overapproximation of* mods_φ *and let* aconfs_φ *be an underapproximation of* confs_φ*.*
1) *If* $\mathsf{gfp}\, \mathit{amods}_\varphi = \emptyset$ *then* φ *is unsatisfiable.*
2) *If* $\mathsf{lfp}\, \mathit{aconfs}_\varphi = \mathit{Structs}$ *then* φ *is unsatisfiable.*

One view is that overapproximations of mods_φ perform deduction by establishing necessary properties of models, while underapproximations of confs_φ perform abduction by finding sufficient conditions (or explanations) for conflicts.

III. Lifting CDCL to Abstractions

CDCL consists of two interacting phases, model search and conflict analysis. Model search aims to find satisfying assignments for the formula. This process may fail and encounter a *conflicting* partial assignment, that is, a partial assignment that contains only countermodels. Conflict analysis extracts a general reason which is used to derive a new lemma over the search space in the form of a clause. In this section we show how model search and conflict analysis can be lifted to abstractions to yield an Abstract CDCL (ACDCL) algorithm. We assume familiarity with CDCL [14].

A. Abstract Model Search

Model search alternates two steps, *deductions* and *decisions*, which refine a given partial assignment. We model these steps as transformers on abstract lattices.

1) Deduction: Deduction rules are overapproximations of the model transformer. Modern CDCL solvers use efficient but imprecise overapproximations, such as the unit rule.

Definition III.1. A *deduction rule* for an abstraction Q of $\wp(\mathit{Structs})$ and formula $\varphi \in \mathit{Forms}$ is an overapproximation $\mathit{ded} : Q \to Q$ of mods_φ.

Example III.1. Consider the formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ with $\varphi_1 \mathrel{\hat=} (5 \leq x \leq 10)$, $\varphi_2 \mathrel{\hat=} (x = y)$ and $\varphi_3 \mathrel{\hat=} (y = z)$. We define a deduction rule $\mathit{ded}(S) \mathrel{\hat=} \bigsqcap_{i \in \{1,2,3\}} \zeta(\mathit{mods}_{\varphi_i}(\zeta(S)))$ by computing the best overapproximations of $\mathit{mods}_{\varphi_i}$ for $i \in \{1, 2, 3\}$ and intersecting the result. We now compute the greatest fixed point $\mathsf{gfp}\, \mathit{ded}$ which is the analogue of performing Boolean constraint propagation in propositional solvers, where $F_0 = \top$ and $F_i = \mathit{ded}(F_{i-1})$.

$$\begin{aligned} &F_0 = \mathit{FloatAsg} && F_1 = \langle x : [5.0, 10.0]\rangle \\ &F_2 = F_1 \sqcap \langle y : [5.0, 10.0]\rangle && F_3 = F_2 \sqcap \langle z : [5.0, 10.0]\rangle \end{aligned}$$

The resulting element $F_3 = \langle x : [5.0, 10.0], y : [5.0, 10.0], z : [5.0, 10.0]\rangle$ imprecisely overapproximates the set of models.

2) Decisions: Once no new information can be deduced, a CDCL solver makes a decision by restricting the value of a proposition p to a truth value v. In lattice theoretic terms, this can be viewed as computation of the greatest lower bound $\pi \sqcap \langle p : v\rangle$, where π is the original partial assignment.

In terms of the abstraction, an important property of singleton partial assignments is that their complement is precisely expressible as a partial assignment. We generalise:

Definition III.2. Let Q be an abstraction of $\wp(\mathit{Structs})$. The set of *complementable elements* $\mathit{Comp}(Q) \subseteq Q$ is the set of all $q \in Q$ such that $\overline{q} = \mathit{Structs} \setminus q$ is also in Q. A transformer $f : Q \to Q$ is *complementable* if $\mathit{Img}(f) \subseteq \mathit{Comp}(Q)$.

Example III.2 (Complementable elements)**.** Complementable interval assignments map a variable to a half-open interval. The element $\sigma = \langle x : [+0, \max_{\preceq}(\mathbb{F})]\rangle$ is complementable, the elements $\langle x{:}[1, 2]\rangle$ and $\langle x{:}[1, \max_{\preceq}(\mathbb{F})], y{:}[4.2, \max_{\preceq}(\mathbb{F})]\rangle$ are not.

For convenience, we write complementable elements $\langle x{:}[c, \max_{\preceq}(\mathbb{F})]\rangle$ and $\langle x{:}[\min_{\preceq}(\mathbb{F}), c]\rangle$ as $\langle x \succeq c\rangle$ and $\langle x \preceq c\rangle$, respectively.

Modern CDCL solvers implement decision heuristics that use statistical information generated from the execution history of the procedure. Since we do not intend to give a fully stateful account of CDCL here, we abstractly formalise this idea by defining $\mathbb{H}$ to be a set of *execution histories*.

Definition III.3. A *decision heuristic* for an abstraction Q of $\wp(\mathit{Structs})$ and $\varphi \in \mathit{Forms}$ is a function $\mathit{decide} : \mathbb{H} \to Q \to Q$ s.t. for all $h \in \mathbb{H}, q \in Q$, $\mathit{decide}(h)(q)$ is a complementable element and $q \sqcap d = q$ implies that q is a set of models of φ.

B. Abstract Conflict Analysis

Model search iterates deduction and decisions until a conflicting element a is encountered, that is, an element that does not represent any models. The aim of conflict analysis is to obtain a more general element $a' \supseteq a$ that is still conflicting. Conflict analysis can be viewed as an instance of abductive reasoning, since the goal is to find a general reason or explanation for a given deduction.

1) Abduction: A conflict analysis procedure computes a *propositional abduction rule*, which generalises explanations for a deduction π over a formula φ. We model this as a transformer $\mathit{abd}_{\varphi,\pi} : \mathit{PartAsg} \to \mathit{PartAsg}$ such that for any model $\sigma \in \mathit{abd}_{\varphi,\pi}(\pi')$ of φ, σ is in π' or in π. In other words, the transformer may only introduce models that are in π. We generalise:

Definition III.4. An *abduction rule* for an abstraction Q of $\wp(\mathit{Structs})$, an element $q \in Q$ and a formula $\varphi \in \mathit{Forms}$ is an extensive underapproximation $\mathit{abd}_{\varphi,q} : Q \to Q$ of the transformer $\lambda x.\ \mathit{confs}_{\varphi}(x) \cup q$.

Essentially, one could simply work with underapproximations of confs_{φ}. The slight variation presented above allows to find explanations not only for conflicts, but also for specific deductions q.

2) Choice: Note that in contrast to deduction rules, which are overapproximations, there is no single best underapproximate abduction rule. In general, multiple maximally general abduction rules of incomparable generality may exist.

Example III.3. Let $\varphi = \ldots \wedge (p \vee q) \wedge (p \vee \neg q)$ be a propositional CNF formula, and assume that the partial assignment $\pi = \langle p : 0, q : 1\rangle$ is conflicting with φ. The presence of either assignment to p or q is sufficient to deduce the other. We can build two incomparable abduction transformers abd^1 and abd^2 with $\mathit{abd}^1_{\varphi,\bot}(\pi) = \langle p{:}0\rangle$ and $\mathit{abd}^2_{\varphi,\bot}(\pi) = \langle q{:}1\rangle$.

Example III.4. Let $\varphi = x + y \leq 10.0$ be an FPA formula. The interval assignment $\sigma = \langle x < 10.0, y \preceq 10.0\rangle$ is conflicting w.r.t. φ, since $x + y$ is at least 20.0. We can build two incomparable abduction transformers, abd^1 and abd^2 with $\mathit{abd}^1_{\varphi,\bot}(\sigma) = \langle x \preceq 10.0, y \preceq 0.0\rangle$ and $\mathit{abd}^2_{\varphi,\bot}(\sigma) = \langle x \preceq 1.0, y \preceq 9.0\rangle$.

The abduction rule used in propositional CDCL solvers is computed using a graph-based algorithm which will be discussed in more detail in the next section. The absence of a best abduction operator is reflected by the possibility of extracting various incomparable partial assignments from a single graph. Among these, one is heuristically chosen. We formalise this heuristic choice as a function that takes as argument an execution history and returns an abduction rule.

Definition III.5. A *choice heuristic* for an abstraction Q of $\wp(\mathit{Structs})$, $q \in Q$ and $\varphi \in \mathit{Forms}$ is a function $\mathit{choose}_{q,\varphi} : \mathbb{H} \to Q \to Q$ s.t. for all $h \in \mathbb{H}$, $\mathit{choose}_{\varphi,q}(h)$ is an abduction rule for q and φ.

IV. Learning in Abstract Implication Graphs

Effective learning is essential for the performance of CDCL. Learning algorithms in CDCL solvers operate over an implication graph, a data structure that records decisions and the result of deductions. We present a generalisation to *abstract* implication graphs. There are various aspects of the CDCL framework that we do not discuss here, such as restarts, backjumps and learning of asserting clauses. These can also be lifted from the propositional case in a relatively straightforward way.

A. Abstract Trails from Complementable Decompositions

Partial assignments and intervals share an important property regarding the decomposition of lattice elements.

Example IV.1. Let $\pi = \langle x{:}1, y{:}0, z{:}1\rangle$ be a partial assignment. The element π is not complementable, but it can be decomposed into $\pi = \langle x{:}1\rangle \sqcap \langle y{:}0\rangle \sqcap \langle z : l\rangle$. Each element of the above decomposition is complementable.

Now let $\sigma = \langle x : [0.5, 2.2], y : [0.5, \max_{\preceq}(\mathbb{F})]\rangle$ be an interval assignment. Analogous to the previous case, the complement of σ is not an interval assignment, but σ can be decomposed into complementable elements as $\langle x \succeq 0.5\rangle \sqcap \langle x \preceq 2.2\rangle \sqcap \langle y \succeq 0.5\rangle$.

Definition IV.1. An abstraction Q of $\wp(\mathit{Structs})$ has *complementable decompositions* if for every element q of Q there is a finite set $S \subseteq Comp(Q)$ of complementable elements such that $q = \bigsqcap S$.

As illustrated above, both partial assignments and interval assignments admit complementable decompositions. We assume the existence of a decomposition function $\mathsf{decomp} : Q \to \wp(Comp(Q))$. Implication graph construction necessitates a decompositon of deduction rules into complementable transformers.

Example IV.2. Let ded be the best deduction rule over interval assignments for the predicate $-x = y$, and let $\sigma = \langle x : [5.0, 10.0]\rangle$. It holds that $ded(\sigma) = \sigma \sqcap \langle y : [-10.0, -5.0]\rangle$. We can decompose ded into a set of complementable rules $Ded = \{ded^l_x, ded^u_x, ded^l_y, ded^u_y\}$ s.t. $\bigsqcap Ded = ded$, and each of the elements of Ded infers a lower or an upper bound on x or y: $ded^l_x(\sigma) = \langle x \succeq 5.0\rangle$, $ded^u_x(\sigma) = \langle x \preceq 10.0\rangle$, $ded^l_y(\sigma) = \langle y \succeq -10.0\rangle$ and $ded^u_y(\sigma) = \langle y \preceq -5.0\rangle$.

Abstract Trail: CDCL solvers record decisions and deductions in a stack-based data structure called *trail*, which records variable assignments due to decisions and deductions. Deductions are associated with the clause used to derive them.

An *abstract trail* is a finite sequence of complementable elements in $Comp(Q)$. We denote the i-th element of a trail tr by tr_i, the concatenation of two sequences tr, tr' by $tr \cdot tr'$ and the subsequence $tr_i \ldots tr_j$ by $tr_{i:j}$. In Algorithm 1, we give a generic model search procedure that extends an abstract trail tr and maps trail indices to reasons in a map $reasons$. The procedure can be instantiated over any abstraction Q with complementable decompositions and a decomposition Ded of the deduction rule into complementable rules.

```
modelSearch($tr$, $reasons$, $Ded$)
    loop
        repeat
            forall the $ded \in Ded$ do
                $q \leftarrow ded(\bigsqcap tr)$;
                if $(\bigsqcap tr) \sqcap q \sqsubset \bigsqcap tr$ then
                    $tr \leftarrow tr \cdot q$;
                    $reasons[\,|tr|\,] \leftarrow ded$;
                end
                if $q = \bot$ then return $(tr, reasons)$
            end
        until $tr$ unchanged;
        $q \leftarrow decide(\mathsf{getHistory}())(\bigsqcap tr)$;
        if $q \not\sqsubset \bigsqcap tr$ then return SAT ;
        $tr \leftarrow tr \cdot q$;
```

Algorithm 1: Model search with Abstract Trail

The current abstract element is represented by the greatest lower bound $\bigsqcap tr$. Deduction iterates over all deduction rules $ded \in D$, and appends a new element to the abstract trail if applying ded refines $\bigsqcap tr$. If a conflict is deduced, the procedure returns the trail and the reason map. This process is iterated until no new deductions can be made, at which point a decision is attempted. If the current element cannot be refined further, SAT is returned, otherwise the procedure appends the decision to the trail and reenters the deduction phase.

B. FIRST-UIP *in Abstract Conflict Graphs*

In propositional CDCL, a trail implicitly encodes a graph structure that records dependencies between deductions on the trail. The edges are represented implicitly by the clauses associated with each element. The FIRST-UIP algorithm [15] is a popular strategy for learning: it is a strategy to choose a set of nodes in this graph called a *cut* that suffices to produce a conflict. We now give a generalisation of FIRST-UIP to abstractions. Naively lifting the algorithm is insufficient to learn good reasons in the interval abstraction as the following example will illustrate.

Example IV.3. Consider the FPA formula $z = y \wedge x = y \cdot z \wedge x < 0$ and the interval assignment $\sigma = \langle z \preceq -5.0\rangle$. Starting from σ, we can make the following deductions.

$$\begin{tikzcd}
\langle z \preceq -5.0\rangle \arrow[rr] \arrow[dr] & & \langle x \succeq 25.0\rangle \arrow[r] & \bot \\
& \langle y \preceq -5.0\rangle \arrow[ur] & &
\end{tikzcd}$$

Arrows indicate sufficient conditions for deduction, e.g., $\langle x \succeq 25.0\rangle$ can be deduced from the conjunction of $\langle z \preceq -5.0\rangle$ and $\langle y \preceq -5.0\rangle$. The last deduction $\langle x \succeq 25.0\rangle$ conflicts with the constraint $x < 0$. A classic conflict cutting algorithm may analyse the above graph to conclude that $\pi = \langle z \preceq -5.0\rangle$ is the reason for the conflict. It is easy to see though that there is a much more general reason: The conflict can be deduced in this way whenever z is negative.

```
analyse($tr$, $reasons$)
    $i \leftarrow |tr|$; $m \leftarrow \{1 \mapsto \top, \ldots, (i-1) \mapsto \top, i \mapsto \bot\}$;
    loop
        $q \leftarrow$ generalise($\bigsqcap tr_{1:i}$, $reasons[i]$, $m[i]$);
        updateMarking($q, tr, m$);
        $m[i] \leftarrow \top$; $i \leftarrow i - 1$;
        if open($tr, m$) $= 1$ then
            return $\bigsqcap_{1 \le i \le |tr|} m[i]$;
        end

generalise($q, d, r$)
    repeat
        $abd \leftarrow choose_{d,r}(\mathsf{getHistory}())$;
        $q \leftarrow abd(q)$;
    until $q$ unchanged;
    return $q$;
updateMarking($q, tr, m$)
    $\Pi \leftarrow \mathsf{decomp}(q)$;
    forall the $c$ in $\Pi$ do
        $r \leftarrow$ smallest index $r'$ s.t. $tr_{r'} \sqsubseteq c$;
        $m[r] \leftarrow m[r] \sqcap c$;
    end
```

Algorithm 2: Abstract FIRST-UIP

Abstract FIRST-UIP breaks down the global abduction task of conflict analysis by finding generalised explanations for single deduction results. We associate with each $ded \in Ded$ a separate choice function $choose_{ded,q}$, which maps an execution history h to an abductive transformer for inferring q.

The procedure is presented in Algorithm 2. It takes as input a conflicting trail tr with final element $\bot$ and a mapping from indices i to the deduction rule used to derive an element tr_i. The main data structure is a *marking* m which maps trail indices to elements of $Comp(Q)$. Essentially, m maps each element of

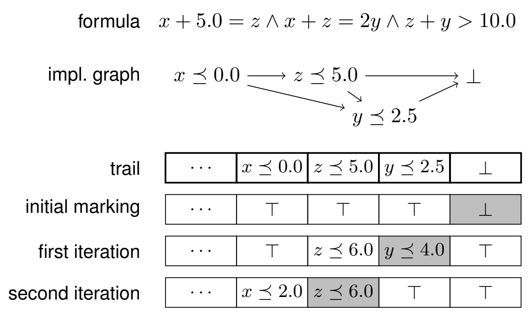

formula $x + 5.0 = z \wedge x + z = 2y \wedge z + y > 10.0$
impl. graph $x \preceq 0.0$ $z \preceq 5.0$ $\bot$ $y \preceq 2.5$
trail $\cdots$ $x \preceq 0.0$ $z \preceq 5.0$ $y \preceq 2.5$ $\bot$
initial marking $\cdots$ $\top$ $\top$ $\top$ $\bot$
first iteration $\cdots$ $\top$ $z \preceq 6.0$ $y \preceq 4.0$ $\top$
second iteration $\cdots$ $x \preceq 2.0$ $z \preceq 6.0$ $\top$ $\top$

Fig. 1. Markings in Abstract FIRST-UIP

the trail tr to a generalisation that is still sufficient to produce a conflict.

Initially, m maps only the final, conflicting element to $\bot$ and everything else to $\top$. The procedure steps backwards through the trail. A call to a function generalise(q,d,r) finds a generalisation of $q \in Q$ such that the current trail marking r can still be deduced. This is done by computing a fixed point using heuristic choice over abductive transformers. The generalised deduction reason is decomposed into its complementables, and for each element c of the decomposition, the earliest occurrence of a stronger element on the trail is marked with c. Finally, the current marking is removed and the algorithm proceeds.

An example execution of the algorithm is illustrated in Figure 1. There, an implication graph and corresponding trail is shown which records consequences of a decision $x \preceq 0.0$. Similar to propositional CDCL, no explicit graph is constructed. Instead, the algorithm implicitly explores the graph via markings, which overapproximate the trail pointwise and encode sufficient conditions for unsatisfiability. The first iteration of the algorithm determines via abduction that $\bot$ can be ensured whenever $z \preceq 6.0$ and $y \preceq 4.0$ are the case. The second iteration finds that $y \preceq 4.0$ can be dropped from the reason if $x \preceq 2.0$ holds in addition to $z \preceq 6.0$.

It is an invariant during the run of the procedure that the greatest lower bound over all markings is sufficient to ensure a conflict. Hence the procedure could essentially terminate during any iteration and yield a sound global abduction result. We use the usual FIRST-UIP termination criterion and return once the number of open paths $\mathrm{open}(tr, m)$ reaches 1. This number is defined as the number of indices j greater or equal to the index of the most recent decision, such that $m[j] \neq \top$.

C. Abstract Clause Learning

Propositional solvers learn new clauses that express the negation of the conflict analysis result. The new clauses open up further possibilities for deduction using the unit rule. The unit rule states that for a clause $l_1 \vee \ldots \vee l_k$, if l_1 to l_{k-1} are contradicted by the current partial assignment, then the partial assignment can be refined to make l_k evaluate to true.

We model learning directly as learning of a new deduction rule, rather than learning a formula in the logic. A lattice-theoretic generalisation of the unit rule is given below. Note that we define the rule directly in terms of the conflicting element, rather than its negation.

Definition IV.2. For an abstraction P of $\wp(\mathit{Structs})$ with complementable decompositions, let $c \in P$ be an element that contains no models of φ. The *abstract unit rule* $\mathit{Unit}_c : P \to P$ is defined as follows.

$$\mathit{Unit}_c(p) \mathrel{\hat{=}} \begin{cases} \bot & \text{if } p \sqsubseteq c \\ \overline{r} & \text{otherwise, if } r \in \mathsf{decomp}(c) \text{ and} \\ & \forall r' \in \mathsf{decomp}(c) \setminus \{r\}.\ p \sqsubseteq r' \\ \top & \text{otherwise} \end{cases}$$

Example IV.4. Let $c = \langle x{:}[0.0, 10.0], y \preceq 3.2\rangle$ be a conflicting element of φ. Let $p = \langle x{:}[3.0, 4.0], y{:}[1.0, 1.0]\rangle$, then $\mathit{Unit}_c(p) = \bot$, since $p \sqsubseteq c$. Let $p' = \langle x{:}[3.0, 4.0]\rangle$, then $\mathit{Unit}_c(p') = \langle y \succ 3.2\rangle$, since $p' \sqsubseteq \langle x \succeq 0.0\rangle$ and $p' \sqsubseteq \langle x \preceq 10.0\rangle$.

The unit rule Unit_c for a conflicting element c soundly overapproximates the model transformer. Furthermore, it is complementable; we can perform learning by adding Unit_c to Ded.

V. Implementation and Experiments

We have implemented our approach over floating-point intervals inside the MATHSAT5 SMT solver [16]. We call our prototype tool FP-ACDCL. The implementation uses the MATHSAT5 infrastructure, but is currently independent of its DPLL(T) framework. The implementation provides a generic, abstract CDCL framework with FIRST-UIP learning. The overall architecture is shown in Figure 2. An instantiation requires abstraction-specific implementations of the components described earlier, including deduction, decision making, abduction and heuristic choice. We first elaborate on those aspects of the implementation and then report experimental results.

A. Abstract CDCL for Floating-Point Intervals

1) Deductions: We implement the deduction rule ded using standard Interval Constraint Propagation (ICP) techniques for floating-point numbers, defined e.g., in [2], [17]. The implementation operates on CNF formulae over floating-point predicates.

Propagation is performed using an occurrence-list approach, which associates with each variable a list of the FPA clauses in which the variable occurs. Learnt clauses (corresponding

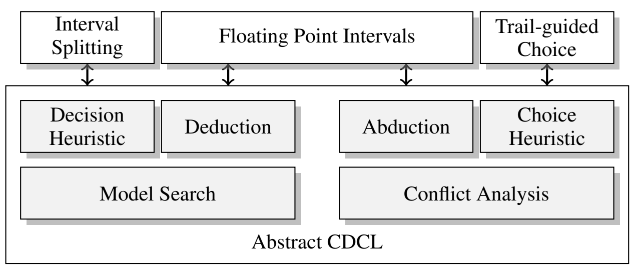


Fig. 2. FP-ACDCL Solver Architecture

to new unit rules) are stored as vectors of complementable elements and are propagated in a similar way. When a deduction is made, we scan the list of affected clauses to check for new deductions to be added to the trail. This is done by applying ICP projection functions to the floating-point predicates in a way that combines purely propositional with theory-specific reasoning. A predicate is conflicting if some variable is assigned the empty interval during ICP. If all predicates of a clause are contradicting, then we have found a conflict with the current interval assignment and ded returns $\bot$. If all but one predicate in a clause are conflicting, then the result of applying ICP to the remaining predicate is the deduction result. In this case, ded returns a list containing one complementable element $\langle x \succeq b\rangle$ (or $\langle x \preceq b\rangle$) for each new bound inferred.

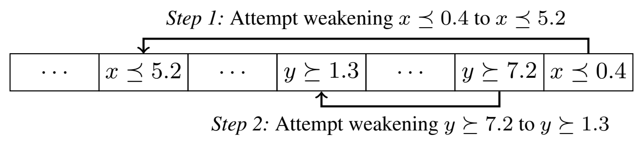
2) Decisions: FP-ACDCL performs decisions by adding to the trail one complementable element $\langle x \succeq b\rangle$ or $\langle x \preceq b\rangle$ that does not contradict the previous value of x. Clearly, there are many possible choices for (i) how to select the variable x, (ii) how to select the bound b, and (iii) how to choose between $\langle x \succeq b\rangle$ and $\langle x \preceq b\rangle$.

In propositional CDCL, each variable can be assigned at most once. In our lifting, a variable can be assigned multiple times with increasingly precise bounds. We have found some level of *fairness* to be critical for performance. Decisions should be balanced across different variables and upper and lower bounds. A strategy that proceeds in a "depth-first" manner, in which the same variable is refined using decisions until it has a singleton value, shows inferior performance compared to a "breadth-first" exploration, in which intervals of all the variables are restricted uniformly. We interpret this finding as indication that the value of abstraction lies in the fact that the search can be guided effectively using general, high-level reasoning, before considering very specific cases.

FP-ACDCL currently performs decisions as follows: (i) variables are statically ordered, and the selection on which variable x to branch is cyclic across this order; (ii) the bound b is chosen to be an approximation of the arithmetic average between the current bounds l and u on x; note that the arithmetic average is different from the median, since floating-point values are unevenly distributed; (iii) the choice between $\langle x \succeq b\rangle$ and $\langle x \preceq b\rangle$ is random. Considering the advances in heuristics for propositional SAT, there is likely a lot of room for enhancing this. In particular, the integration of fairness considerations with activity-based heuristics typically used in modern CDCL solvers could lead to similar performance improvements. This is part of ongoing and future work.

3) Generalised Explanations for Conflict Analysis: In abduction, a trade-off must be made between finding reasons quickly and finding very general reasons. We perform abduction that relaxes bounds iteratively. As mentioned earlier, there may be many incomparable relaxations. Our experiments suggest that the precise way in which bounds are relaxed is extremely important for performance. Fairness considerations similar to those mentioned for the decision heuristic need to be taken into account. However, there is an additional, important criterion. Learnt lemmas are used to drive backjumping. It is therefore preferable to learn deduction rules that allow for backjumping higher in the trail. This will lead to propagations that are affected by a smaller number of decisions, and thus will hold for a larger portion of the search space.

Our choice heuristic, called *trail-guided choice*, is abstraction-independent, and is both fair and aims to increase backjump potential. In the first step, we remove all bounds over variables from the initial reason q which are irrelevant to the deduction. Then we step backwards through the trail and attempt to weaken the current element q using trail elements. The process is illustrated below.


Step 1: Attempt weakening $x \preceq 0.4$ to $x \preceq 5.2$
$\cdots$
$x \preceq 5.2$
$\cdots$
$y \succeq 1.3$
$\cdots$
$y \succeq 7.2$
$x \preceq 0.4$
Step 2: Attempt weakening $y \succeq 7.2$ to $y \succeq 1.3$

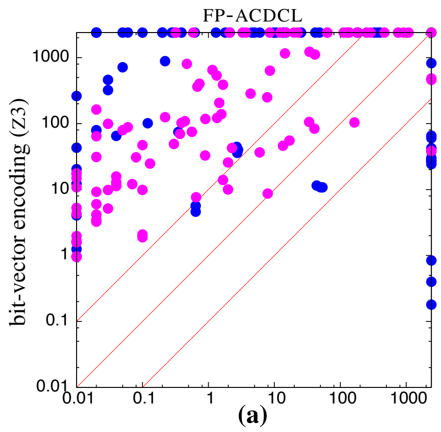
When an element tr_j is encountered such that tr_j is used in q (that is, $q \sqsubseteq tr_j$), we attempt to weaken q by replacing the bound tr_j with the most recent trail element more general than tr_j. If no such element exists, we attempt removing the relevant bound altogether. We check whether the weakened q is still sufficiently strong to deduce d. If not, we undo the weakening, and do not consider any further weakenings with elements more general than tr_j. After this, we repeat the process for element tr_{j-1}. The algorithm terminates once no further generalisations are possible.

Since we step backwards in order of deduction, we heuristically increase the potential for backjumps: The procedure never weakens a bound that was introduced early during model search at the expense of having to uphold a bound that is ensured only at a deep level of the search.

We have experimented with stronger but computationally more expensive generalisation techniques such as finding maximal bounds for deductions by search over floating-point values. Our experiments indicate that the cheaper technique described above is more effective overall. We see two main avenues for improvement: First, for many deductions it is possible to implement good or optimal abduction transformers effectively without search. Second, we expect that dynamic heuristics that take into account statistical information may guide conflict analysis towards useful clauses.

B. Experimental Evaluation

We have evaluated our prototype FP-ACDCL tool over a set of more than 200 benchmark formulas, both satisfiable and unsatisfiable. The formulas have been generated from problems that check (i) ranges on numerical variables and expressions, (ii) error bounds on some numerical computations using different orders of evaluation of subexpressions, and (iii) feasibility of systems of inequalities over bounded floating-point variables. The first two sets originate from verification problems on some C programs performing numerical computations, whereas the instances in the third set are randomly generated. We make our benchmarks and the FP-ACDCL tool available for experimentation by other researchers at http://www.cprover.



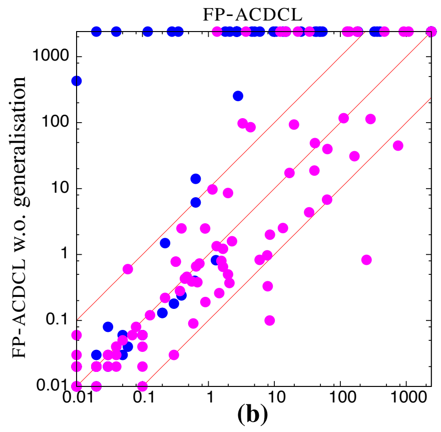


Fig. 3. Comparison of FP-ACDCL against Z3 with bit-vector encoding (a); effects of generalisations in conflict analysis (b). Darker colour indicates unsatisfiability. Points on the borders indicate timeouts (1200 s).

org/fmcad2012/. All results have been obtained on an Intel Xeon machine with 2.6 GHz and 16 GB of memory running Linux, with a time limit of 1200 seconds.

We have performed two different sets of experiments. In the first, we have compared FP-ACDCL with the current state-of-the-art procedures for floating-point arithmetic, based on encoding into bit-vectors. We have generated bit-vector encodings of all the benchmark instances in our set using MATHSAT5 and solved them with the Z3 SMT solver [18], which was the winner of the main bit-vector division in the SMT-COMP 2011 competition. The results of this comparison are reported in Figure 3(a). FP-ACDCL over FPA significantly outperforms Z3 over corresponding bit-vector encodings on most of the instances, often by several orders of magnitude. More specifically, FP-ACDCL could solve 35 benchmarks more than Z3, with an overall total speedup of more than 25x (for the subset of benchmarks that both tools could solve).[1] There are some instances that turn out to be relatively easy for Z3, but cannot be solved by our tool. This is not surprising, since there are simple instances that are not amenable to analysis with ICP, even with the addition of decision-making and learning.[2] To handle such cases, our framework can be instantiated with abstract domains or combinations of domains [13] that are better suited to the problems under analysis.

The second set of experiments is aimed at evaluating the impact of our novel generalisation technique. In order to do this, we have run FP-ACDCL with generalisation of deductions turned off, and compared it with the default FP-ACDCL. Essentially, FP-ACDCL without generalisation corresponds to a naive lifting of the conflict analysis algorithm. The results are summarised in Figure 3(b). From the plot, we can clearly see that generalisation is crucial for the performance of FP-ACDCL: without it, the tool times out in 44 more cases, whereas there is no instance that can be solved only without generalisation. However, there are a number of instances for which performance degrades when using generalisations, sometimes significantly. This can be explained by observing that (i) generalisations come at a runtime cost, which can sometimes induce a non-negligible overhead; (ii) the performance degradation occurs on satisfiable instances (shown in a lighter colour in the plots), for which it is known that the behaviour of CDCL-based approaches is typically unstable (even in the propositional case).

VI. A Survey of Related Work

We separately survey work in three related branches of research: 1) the analysis of floating-point computations, 2) lifting existing decision procedure architectures to richer problem domains and 3) automatic and intelligent precision refinement of abstract analyses.

A. Reasoning about Floating-Point Numbers

This section briefly surveys work in interactive theorem proving, abstract interpretation and decision procedures that target floating-point problems. For a discussion of the special difficulties that arise in this area, see [19].

Theorem Proving: Various floating-point axiomatisations and libraries for interactive theorem provers exist [20]–[23]. Theorem provers have been applied extensively to proving properties over floating-point algorithms or hardware [24]–[31]. While theorem proving approaches have the potential to be sound and complete, they require substantial manual work, although sophisticated (but incomplete) strategies exist to automate substeps of the proof, e.g., [32]. A preliminary attempt to integrate such techniques with SMT solvers has recently been proposed in [33].

Abstract Interpretation: Analysis of floating-point computations has also been extensively studied in abstract interpretation. An approach to specifying floating-point properties over programs was proposed in [34]. A number of general purpose abstract domains have been constructed for the analysis of floating-point programs [35]–[40]. In addition, specialised approaches exist which target specific problem domains such

[1]FP-ACDCL timed out in 28 instances, whereas Z3 ran out of time or memory in 63 cases. On the subset of benchmarks solved by both tools, the total run time was of 585 seconds for FP-ACDCL, and of 15973 seconds for Z3.

[2]A simple example of this is the formula $x = y \wedge x \neq y$, which requires an abstraction that can express relationships between variables. Intervals are insufficient to efficiently solve this problem.

as numerical filters [41], [42]. The approaches discussed so far mainly aim at establishing the result of a floating-point computation. An orthogonal line of research is to analyse the deviation of a floating-point computation from its real counterpart by studying the propagation of rounding errors [43], [44]. Case studies for this approach are given in [45], [46]. Abstract interpretation techniques provide a soundness guarantee, but may yield imprecise results.

Decision Procedures: In the area of decision procedures, study of floating-point problems is relatively scarce. Work in constraint programming [47] shows how approximation with real numbers can be used to soundly restrict the scope of floating-point values. In [17], a symbolic execution approach for floating-point problems is presented, which combines interval propagation with explicit search for satisfiable floating-point assignments. An SMTLIB theory of FPA was presented in [12]. Recent decision procedures for floating-point logic are based on propositional encodings of floating-point constraints. Examples of this approach are implemented in MATHSAT5 [16], CBMC [48] and Sonolar [49]. A difficulty of this approach is that even simple floating-point formulas can have extremely large propositional encodings, which can be hard for current SAT solvers. This problem is addressed in [1], which uses a combination of over- and underapproximate propositional abstractions in order to keep the size of the search space as small as possible.

B. Lifting Decision Procedures

The practical success of CDCL solvers has given rise to various attempts to lift the algorithmic core of CDCL to new problem domains. This idea is extensively studied in the field of satisfiability modulo theories. The most popular such lifting is the DPLL(T) framework [50], which separates theory-specific reasoning from Boolean reasoning over the structure of the formula. Typically a propositional CDCL solver is used to reason about the Boolean structure while an ad-hoc procedure is used for theory reasoning. The DPLL(T) framework can suffer from some difficulties that arise from this separation. To alleviate these problems, approaches such as *theory decisions on demand* [4] and theory-based decision heuristics [51] have been proposed.

Our work is co-located in the context of natural-domain SMT [5], which aims to lift steps of the CDCL algorithm to operate directly over the theory. Notable examples of such approaches have been presented for equality logic with uninterpreted functions [52], linear real arithmetic and difference logic [5], [6], linear integer arithmetic [7], nonlinear integer arithmetic [9], and nonlinear real arithmetic [8]. The work in [9] is most similar to ours since it also operates over intervals and uses an implication graph construction.

We follow a slightly different approach to generalisation based on abstract interpretation. The work in [10] shows that SAT solvers can naturally be considered as abstract interpreters for logical formulas. Generalisations can then be obtained by using different abstract domains. Our work is an application of this insight. A similar line of research was independently undertaken in [53], [54], which presents an abstract-interpretation based generalisation of Stålmarck's method and an application to computation of abstract transformers.

C. Refining Abstract Analyses

A number of program analyses exist that use decision procedures or decision procedure architectures to refine a base analysis. A lifting of CDCL to program analyses over abstract domains is given in [55]. In [56], a decision-procedure based software model checker is presented that imitates the architecture of a CDCL solver. A lifting of DPLL(T) to refinement of abstract analyses is presented in [57] which combines a CDCL solver with an abstract interpreter.

Modern CDCL solvers can be viewed as refinements of the original DPLL algorithm [58], which is based on case-analysis. Case analysis has been studied in the abstract interpretation literature. The formal basis is given by cardinal power domains, already discussed in [13], in which a base domain is refined with a lattice of cases. The framework of *trace partitioning* [59] describes a systematic refinement framework for programs based on case analysis. The DPLL algorithm can be viewed as a special instance of dynamic trace partitioning applied to the analysis of logical formulas.

VII. Conclusions and Future Work

We have presented a decision procedure for the theory of floating-point arithmetic based on a strict lifting of the conflict analysis algorithm used in modern CDCL solvers to abstract domains. We have shown that, for a certain class of formulas, this approach significantly outperforms current complete solvers based on bit-vector encodings. Both our formalism and our implementation are modular and separate the CDCL algorithm from the details of the underlying abstraction. Furthermore, the overall architecture is not tied to analysing properties over floating-point formulas.

We are interested in a number of avenues of future research. One of these is a comparison of abstract CDCL and DPLL(T)-based architectures, and investigating possible integrations. Another avenue of research is instantiating ACDCL with richer abstractions (e.g., octagons). Combination and refinements of abstractions are well studied in the abstract interpretation literature [13]. Recent work [60] has shown that Nelson-Oppen theory combination is an instance of a product construction over abstract domains. We hope to apply this work to obtain effective theory combination within ACDCL. In addition, product constructions can be used to enhance the reasoning capabilities within a single theory, e.g., by fusing interval-based reasoning over floating-point numbers and propositional reasoning about the corresponding bit-vector encoding.

We see this work as a step towards integrating the abstract interpretation point of view with algorithmic advances made in the area of decision procedures. Black-box frameworks such as DPLL(T) abstract away from the details of their component procedures. Abstract interpretation can be used to express an orthogonal, algebraic "white-box" view which, we believe, has uses in both theory and practice.

Acknowledgements: We gratefully acknowledge the contributions of Vijay D'Silva who helped create the formal framework advocated in this paper.

References

[1] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *FMCAD*. IEEE, 2009, pp. 69–76.

[2] C. Michel, "Exact projection functions for floating point number constraints," in *AMAI*, 2002.

[3] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 825–885.

[4] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Splitting on demand in SAT modulo theories," in *LPAR*, 2006, pp. 512–526.

[5] S. Cotton, "Natural domain SMT: a preliminary assessment," in *FORMATS*. Springer, 2010, pp. 77–91.

[6] K. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to richer logics," in *CAV*. Springer, 2009, pp. 462–476.

[7] D. Jovanovic and L. de Moura, "Cutting to the Chase: Solving Linear Integer Arithmetic," in *CADE*. Springer, 2011, pp. 338–353.

[8] ——, "Solving non-linear arithmetic," in *IJCAR*. Springer, 2012, pp. 339–354.

[9] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure," *JSAT*, vol. 1, no. 3-4, pp. 209–236, 2007.

[10] V. D'Silva, L. Haller, and D. Kroening, "Satisfiability solvers are static analyzers," in *SAS*, ser. LNCS, vol. 7460. Springer, 2012, pp. 317–333.

[11] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *ICCAD*. ACM, 2001, pp. 279–285.

[12] P. Rümmer and T. Wahl, "An SMT-LIB theory of binary floating-point arithmetic," in *SMT Workshop*, 2010.

[13] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *POPL*, 1979, pp. 269–282.

[14] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 131–153.

[15] L. Zhang, C. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *ICCAD*. IEEE, 2001, pp. 279–285.

[16] A. Griggio, "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic," *JSAT*, vol. 8, pp. 1–27, January 2012.

[17] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *STVR.*, vol. 16, no. 2, pp. 97–121, 2006.

[18] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[19] D. Monniaux, "The pitfalls of verifying floating-point computations," *TOPLAS*, vol. 30, no. 3, 2008.

[20] M. Daumas, L. Rideau, and L. Théry, "A generic library for floating-point numbers and its application to exact computing," in *TPHOLs*. Springer, 2001, pp. 169–184.

[21] G. Melquiond, "Floating-point arithmetic in the Coq system," *Inf. Comput.*, vol. 216, pp. 14–23, 2012.

[22] P. S. Miner, "Defining the IEEE-854 floating-point standard in PVS," NASA, Langley Research, PVS. Technical Memorandum 110167, 1995.

[23] J. Harrison, "A machine-checked theory of floating point arithmetic," in *TPHOLs*. Springer, 1999, pp. 113–130.

[24] ——, "Floating point verification in HOL light: The exponential function," *FMSD*, vol. 16, no. 3, pp. 271–305, 2000.

[25] ——, "Formal verification of square root algorithms," *FMSD*, vol. 22, no. 2, pp. 143–153, 2003.

[26] ——, "Floating-point verification," *J. UCS*, vol. 13, no. 5, pp. 629–638, 2007.

[27] B. Akbarpour, A. Abdel-Hamid, S. Tahar, and J. Harrison, "Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL," *Comput. J.*, vol. 53, no. 4, pp. 465–488, 2010.

[28] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," in *TPHOLs*. Springer, 2000, pp. 338–355.

[29] J. S. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm," *TC*, vol. 47, 1996.

[30] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *JCM*, vol. 1, pp. 148–200, 1998.

[31] J. Harrison, "Formal verification of floating point trigonometric functions," in *FMCAD*, 2000, pp. 217–233.

[32] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *IJCAR*. Springer, 2010, pp. 127–141.

[33] S. Conchon, G. Melquiond, C. Roux, and M. Iguernelala, "Built-in Treatment of an Axiomatic Floating-Point Theory for SMT Solvers," in *SMT Workshop*, 2012.

[34] S. Boldo and J. Filliâtre, "Formal verification of floating-point programs," in *ARITH*. IEEE, 2007, pp. 187–194.

[35] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *ESOP*. Springer, 2004, pp. 3–17.

[36] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *APLAS*. Springer, 2008, pp. 3–18.

[37] L. Chen, A. Miné, J. Wang, and P. Cousot, "Interval polyhedra: An abstract domain to infer interval linear relationships," in *SAS*. Springer, 2009, pp. 309–325.

[38] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *CAV*. Springer, 2009, pp. 661–667.

[39] A. Chapoutot, "Interval slopes as a numerical abstract domain for floating-point variables," in *SAS*. Springer, 2010, pp. 184–200.

[40] L. Chen, A. Miné, J. Wang, and P. Cousot, "An abstract domain to discover interval linear equalities," in *VMCAI*. Springer, 2010, pp. 112–128.

[41] J. Feret, "Static analysis of digital filters," in *ESOP*. Springer, 2004, pp. 33–48.

[42] D. Monniaux, "Compositional analysis of floating-point linear numerical filters," in *CAV*. Springer, 2005, pp. 199–212.

[43] E. Goubault, "Static analyses of the precision of floating-point operations," in *SAS*. Springer, 2001, pp. 234–259.

[44] K. Ghorbal, E. Goubault, and S. Putot, "The zonotope abstract domain Taylor1+," in *CAV*. Springer, 2009, pp. 627–633.

[45] E. Goubault, S. Putot, P. Baufreton, and J. Gassino, "Static analysis of the accuracy in control systems: Principles and experiments," in *FMICS*. Springer, 2007, pp. 3–20.

[46] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *FMICS*, 2009, pp. 53–69.

[47] C. Michel, M. Rueher, and Y. Lebbah, "Solving constraints over floating-point numbers," in *CP*, 2001, pp. 524–538.

[48] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[49] E. V. Jan Peleska and F. Lapschies, "Automated test case generation with SMT-solving and abstract interpretation," in *NFM*. Springer, 2011, pp. 298–312.

[50] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *CAV*. Springer, 2004, pp. 175–188.

[51] D. Goldwasser, O. Strichman, and S. Fine, "A theory-based decision heuristic for DPLL(T)," in *FMCAD*, 2008, pp. 1–8.

[52] B. Badban, J. van de Pol, O. Tveretina, and H. Zantema, "Generalizing DPLL and satisfiability for equalities," *Inf. Comput.*, vol. 205, no. 8, pp. 1188–1211, 2007.

[53] A. Thakur and T. Reps, "A method for symbolic computation of abstract operations," in *CAV*. Springer, 2012.

[54] ——, "A generalization of Stålmarck's method," in *SAS*. Springer, 2012.

[55] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig, "Numeric bounds analysis with conflict-driven learning," in *TACAS*. Springer, 2012, pp. 48–63.

[56] K. L. McMillan, "Lazy annotation for program testing and verification," in *CAV*, 2010, pp. 104–118.

[57] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Program analysis via satisfiability modulo path programs," in *POPL*, 2010, pp. 71–82.

[58] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *CACM*, vol. 5, pp. 394–397, July 1962.

[59] X. Rival and L. Mauborgne, "The trace partitioning abstract domain," *TOPLAS*, vol. 29, no. 5, 2007.

[60] P. Cousot, R. Cousot, and L. Mauborgne, "The reduced product of abstract domains and the combination of decision procedures," in *FoSSaCS*, 2011, pp. 456–472.

Formal Verification of Error Correcting Circuits Using Computational Algebraic Geometry

Alexey Lvov, Luis A. Lastras-Montaño
IBM T.J.Watson Research Center
Yorktown Heights, NY, 10598
{lvov, lastrasl}@us.ibm.com

Viresh Paruthi, Robert Shadowen, Ali El-Zein
IBM Systems and Technology Group
Austin, TX, 78758
{vparuthi, shadowen, elzein}@us.ibm.com

Abstract— **Algebraic error correcting codes (ECC) are widely used to implement reliability features in modern servers and systems and pose a formidable verification challenge. We present a novel methodology and techniques for provably correct design of ECC logics. The methodology is comprised of a design specification method that directly exposes the ECC algorithm's underlying math to a verification layer, encapsulated in a tool "BLUEVERI" , which establishes the correctness of the design conclusively by using an apparatus of computational algebraic geometry (Buchberger's algorithm for Gröbner basis construction). We present results from its application to example circuits to demonstrate the effectiveness of the approach. The methodology has been successfully applied to prove correctness of large error correcting circuits on IBM's POWER systems to protect memory storage and processor to memory communication, as well as a host of smaller error correcting circuits.**

I. INTRODUCTION

ECCs are widely used in practice to protect data against random errors that inevitably occur during transmission as well as during prolonged storage. As semiconductor technology is scaling down to the nanometer regime and tens of gigabits per second transmission rates, error-free data handling requires larger and more sophisticated error correcting circuits, with the code construction and encoding/decoding algorithms almost always going beyond the templates found in classical literature due to feature set requirements. For example, the IBM z196 systems feature "RAIM" (Redundant Array of Independent Memory, [1], [2]) with a 90 byte ECC that allows the system to recover instantaneously from a full DIMM failure even in the presence of additional chip failures. Each such error correcting circuit has to be individually designed and programmed by a human designer. The resulting implementation complexity in hardware can lead to design errors which can cause costly re-spins of the Silicon and derail schedules. Establishing correctness/verification of such complex hardware is of critical importance, though poses formidable challenges.

Traditional verification methods such as software simulation, hardware-accelerated simulation or post-Silicon debug offer insufficient coverage given the difficult nature of the logic and the large solution space to be investigated. State-of-the-art formal verification algorithms (which inherently check circuit behavior against all possible legal combinations of inputs) offering high capacity have been found lacking in proving correctness because of their inability to exploit the specifics of the underlying algebra - Galois field arithmetic.

We propose a solution to the problem of complete symbolic verification of logical circuits which substantially rely on arithmetic over Galois fields. Most of the error correcting circuits fall in the above category, as well as some of the circuits for data encryption and arithmetic logic unit (ALU).

The verification technique is encapsulated in a reasoning tool "Blue Code Verifier" - "BLUEVERI" - and applies algebraic geometry methods (e.g. checks on the consistency of polynomial systems of equations using the concept of Gröbner basis and the associated Buchberger's algorithm) to the problem of verifying circuits defined over Galois fields in order to establish correctness of the logic circuit against a mathematical specification. The methodology has been successfully applied to verify real life error correcting codes at IBM resulting in substantially improved verification quality, by providing full proof of the correctness of the design which was otherwise unobtainable, and in improved productivity, via significantly reduced verification time and effort. We expect the improvements to accumulate as the methodology gets applied "out-of-the-box" to future processor chips employing even stronger ECC designs, and will be key to integrate commodity memories in products as well as in the design of communication link transceivers. The techniques involved are applicable to other types of logic circuitry based on Galois field arithmetic such as Elliptic Curve Cryptography.

A. *Previous Art*

Simulation-based methods such as software simulation or hardware-accelerated simulation are inapplicable to the problem of complex ECC verification. This is due to the fact that the problem has large numbers of inputs which precludes an exhaustive exploration to fully verify the ECC circuitry to cover all possible combinations of input bit strings and injected errors (within the claimed error correction capability of the code) and check to see if in each case the decoded bit string is equal to the original one. Directed simulation to cover the vast majority, if not all, of "corner cases" again requires a careful analysis of the code to enumerate correction capability and features - a process which is inherently subject to human limitations and errors. Systematic methods such as SAT or graph-based canonical representations of the logic with Decision Diagrams (DD) such as BDDs [3], BMDs [4], FDDs [5] run out of steam quickly due to the large input space and the complexity of the underlying logic employing exclusive-ORs. Our experience suggests that these existing decision procedures have difficulty scaling to designs beyond circuits with more than 24-bit inputs. Enhanced verification techniques leveraging Transformation-based Verification (TBV) [11] concepts to simplify then prove the designs become capacity gated for 32-bit Galois field algorithms and beyond. Satisfiability Modulo Theory (SMT) solvers which utilize specialized theories to address specific problem domains (e.g. bit-vectors) do not address polynomial equation solving over Galois fields. Our approach addresses this niche and proposes a methodology to solve such systems of polynomial equations over Galois fields efficiently.

A search for verification of Galois field circuits reveals the following applicable references - [6] and [7]. [6] defines a formal first-order logic language for symbolic arithmetic over an arbitrary binary Galois field along with a set of rules for manipulation of formal sentences (such as transformation of the sentence into prenex normal form, usage of DeMorgan's law, elimination of variables etc.). The correctness criterion for parts of some ECC circuits can be formally expressed in this language, e.g. finding the error locator polynomial from the value of the syndrome for Reed-Solomon codes. A formal reasoning in the language is then applied to prove or disprove the correctness statement. The method is only applicable to verification of algorithms which are correct in any $GF(2^k)$ independently of the value of k. In our method the size of the field is specified; in particular this allows the use of constants of the field other than '0' and '1' in the circuit. The method does not employ any of the computational algebraic geometry machinery; that bounds it to purely $GF(2^k)$ circuits (with no bit operations allowed), while our method works on circuits with mixed bit and GF signals (Boolean result of test value operations on GF signals is computed by building Gröbner basis of polynomial algebraic system).

The latter [7] applies Gröbner basis techniques to the very narrow problem of verifying multipliers over a large Galois field. The class of the multipliers is further limited to those based on representation of the large field as an extension of degree m of a smaller field of degree n. The paper reports practical results of verifying multipliers up to maximum field size of $GF(2^{1024}), (m = 32, n = 32)$, but it does not make any attempts to verify circuits other than this multiplier circuit with a fixed structure parameterized with only two integers m and n. In contrast our method is capable of verifying virtually any circuit built with GF, Boolean and mixed operations, with the runtime and memory being the only limiting factors for large circuits.

II. Proposed Method

Our method was first inspired by the need to verify a large 1024-bit input error correction circuit responsible for protecting the memory store as well as the communication between a POWER processor and memory. A traditional

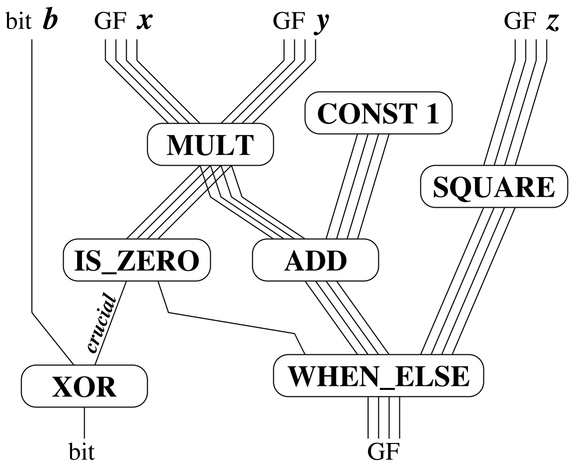


Fig. 1. Example of BLUEVERI circuit representation.

formal verification approach to verify the circuitry quickly became intractable given the vast search space.

The main idea is to use the fact that algebraic ECCs operate mostly on the elements of finite fields, and there are powerful techniques for symbolic reasoning in this domain. The process of verification of such circuits reduces to the verification of a number of algebraic statements of the type "A certain system of multivariate polynomials over a finite field implies some other system of multivariate polynomials over a finite field". The latter problem relates to computational algebraic geometry and can be solved by building Gröbner bases for certain sets of polynomials by using Buchberger's algorithm ([8], pp.77, 82-87).

A. Verification Set-up

The verification set-up consists of two parts: the circuit to be verified, and a check file containing information about the set of legal inputs and the expected values for some set of "crucial" signals; an example of the latter would be an uncorrectable error flag (see subsection III-A) or a signal that tests the equality between two bit vectors (see subsection III-B). The verification task at hand is to formally prove (or disprove) that for any legal combination of inputs, the values of the crucial signals match their expected values.

In a standard processing methodology, the circuit is generally represented by a directed graph where the edges are wires carrying only Boolean signals, and nodes are gates performing only basic Boolean operations. Since we assume that a large portion of the operations in the circuit are operations in $GF(2^k)$ arithmetic, we modify this representation by "glueing" together wires which represent the same $GF(2^k)$ elements and putting "black boxes" around the pieces of the circuit which represent basic $GF(2^k)$ arithmetic operations. Practically this is done by passing a special option to the HDL compiler, telling it to not synthesize functions from a given list. The circuit in our representation typically looks similar to the example on Fig 1.

After this transformation, each wire carries either a Boolean signal or a $GF(2^k)$ signal. For this reason, we generalize the concept of "gate" so that now each gate performs one of the following operations:

- Basic binary arithmetic operations on $GF(2^k)$:
  ADD (both $x+y$ and $x-y$), **MULT** (xy), **DIV** (xy^{2^k-2}).
- Any fixed set of unary operations on $GF(2^k)$ which are linear over $GF(2)$, e.g. Frobenius automorphism (square), projections on elements of a fixed basis, square root, bit permutations etc.
- Any fixed set of $GF(2^k)$ constants (functions without arguments).
- **WHEN_ELSE**(b, x, y) function which returns $GF(2^k)$ element x when bit b is 1 and $GF(2^k)$ element y otherwise.
- $GF(2^k)$ value test functions which return value is a bit: **IS_ZERO**(x), **IS_NONZERO**(x).
- Boolean functions:
  NOT, **AND**, **OR**, **XOR**.

The check file contains algebraic constraints on the $GF(2^k)$ inputs, optionally initial values for some Boolean and $GF(2^k)$ inputs, and the expected values for the crucial Boolean signals testing the desired behavior for the circuit. The crucial signals are restricted to Boolean because any condition on $GF(2^k)$ signals can be expressed as a condition on Boolean signals by adding just a few gates to the circuit. For example, if one wants to state that a GF signal x is equal to a given constant $const$, then one may alternatively assert that we expect

$$\big(\text{IS\_ZERO}(\text{ADD}(x, const))\big)$$

to be equal to 1.

The algebraic constraints are specified in conjunctive normal form (CNF) whose literals are multivariate polynomial equalities or inequalities on the free variables associated with each of the $GF(2^k)$ inputs.

Here is an example of a check file for the circuit on Fig 1:

```
BEGIN_CHECK;

  IN_BITS_SETTINGS;
    b <= '0';

  EXPLICIT_EXPRESSIONS_FOR_SOME_GF_INPUTS;
    x <= "8F3A";

  ALGEBRAIC_CONSTRAINTS_ON_GF_INPUTS;
    [ (y^3 + z^5 == 0) or (y^2 + z != 0) ]
    and
    [ (y == 0) or (z == 0) or (y + z != 0) ]

  BIT_EXPECTED_VALUES;
    crucial must be '1';

END_CHECK;
```

We support multiple checks in one check file in which case our tool verifies them independently one by one, and appending new checks at the end of the file during verification (a necessary feature for the "fork on unresolved bits" mechanism outlined later).

B. Verification Flow

The process starts by assigning a free variable (e.g. the symbolic string identifier used in the HDL file) to each of the $GF(2^k)$ inputs. Next the values of the crucial bit signals are computed one by one by applying the following recursive procedure. The procedures for "... execute the operation ..." will be explained for each type of operation subsequently.

```
COMPUTE_OUTPUT_OF_GATE(signal $g$) {
  // case $g$ is Boolean : Attempt to compute to const. '0' or '1' .
  // case $g$ is $GF(2^k)$ : Compute as a symbolic rational
                            expression in the free variables.
  for all inputs $g_i$ of $g$ {
    COMPUTE_OUTPUT_OF_GATE($g_i$)
  }
  switch (type of $g$) {
    ADD:  ...Execute the operation ...
    MULT: ...Execute the operation ...
    ...   ...
    XOR:  ...Execute the operation ...
  }
}
```

Given unlimited time and memory and assuming that all recursive sub-calls successfully compute values of $g_1, g_2, \ldots$ a call to COMPUTE_OUTPUT_OF_GATE(g) always succeeds if g is a $GF(2^k)$ signal. However, it may fail for Boolean signals because Boolean signals are (generally) not constants but depend on the inputs. If a Boolean signal cannot be computed to '0' or '1' we skip to the next check and add two new checks at the end of the check file assuming values '0' and '1' for that bit by applying the "fork on unresolved bit" procedure described later in this subsection. Note that although it may seem that this would fork on nearly every bit in the circuit, in our experience for ECCs the situation is typically just the opposite: given a restricted set of inputs (e.g. exactly one injected error) most of the Boolean signals in the circuit do not depend on the inputs; an example of this can be seen in subsection III-A in the computation of the uncorrectable error flag of a decoder [1]. Furthermore, BLUEVERI performs signal dependency checks that result in the value of many boolean signals in the circuit not being needed; such booleans never cause a fork as described above.

Given $g_1, g_2, \ldots$, we compute g depending on the type of operation as follows:

ADD and **MULT** : Perform the operation on the mutivariate rational expressions. E.g. ADD$(\frac{x}{y+z}, \frac{y}{x+z}) = \frac{x^2+xz+y^2+yz}{xy+xz+yz+z^2}$, MULT$(x+1, y+1) = xy + x + y + 1$ etc.

UNARY_LINEAR_i : Any operation on $GF(2^k)$ which is linear over $GF(2)$ can be given by a linearized polynomial (a polynomial containing only terms of the form cx^{2^t}, see [9] pp.107-124). Substitute the input rational expression into the linearized polynomial. E.g. in $GF(16)$ $\text{Tr}(x) \overset{\text{def}}{=} x^8 + x^4 + x^2 +$

[1]Very often the uncorrectable error signal is both an internal signal upon which further things depend and also an output by itself.

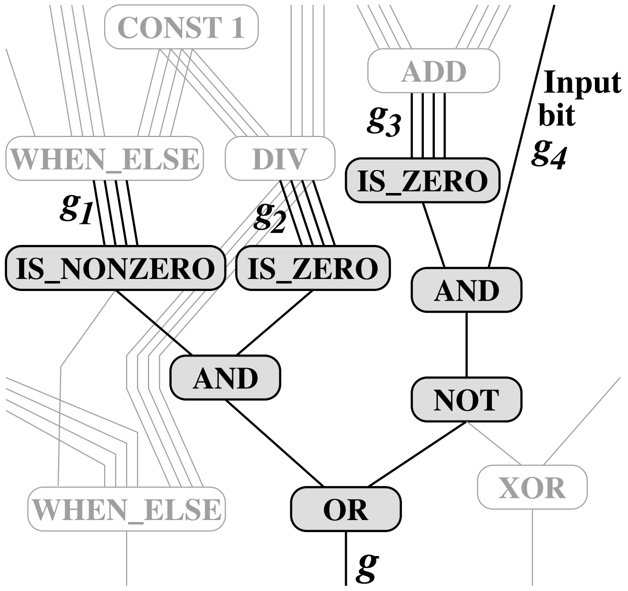


Fig. 2. Example of maximal "algebraic system" subgraph for signal g.

x, $\mathrm{Tr}(y+z^3) = y^8+y^4+y^2+y+z^{24}+z^{12}+z^6+z^3$.

CONST_i : Set signal g to the constant (a rational expression containing no free variables).

WHEN_ELSE(b, X, Y) : Set rational expression g to rational expression X if b is '1' and to rational expression Y otherwise.

IS_ZERO, **IS_NONZERO**, **NOT**, **AND**, **OR**, **XOR** : Computation of values of gates with Boolean output constitutes the most complex part of our algorithm.

To compute the value of g we first find the maximal subgraph consisting of all gates h_j such that there exists a directed path from h_j to g and all gates on this path except for h_j itself are elementary Boolean gates (NOT, AND, OR or XOR). An example is shown on Fig. 2. Note that the subgraph may only contain IS_ZERO, IS_NONZERO and elementary Boolean gates, and any IS_ZERO or IS_NONZERO in the subgraph must be a top most gate. The input signals g_i of the subgraph are either $GF(2^k)$ inputs of value test functions or Boolean inputs of the whole circuit.

By inductive hypothesis for our recursive function COMPUTE_OUTPUT_OF_GATE(g) all $GF(2^k)$-type g_i have already been assigned some rational expression in the free variables, and all Boolean type g_i have been computed to constant '0' or '1' (this is possible for all Boolean inputs to the circuit due to an explicit assignment in the "In bits settings" section of the check which may be set either by the user or as a result of forking on unresolved bits).

The Boolean function given by the subgraph can be written as a conjunctive normal form whose literals are $g_i = 0$ or $g_i \neq 0$, where g_i are rational expressions. As we will show in the description of DIV operation, we always make sure the denominators of our rational expressions cannot be zero. This allows replacement of $g_i = 0$ and $g_i \neq 0$ literals by $\mathrm{numerator}(g_i) = 0$ and $\mathrm{numerator}(g_i) \neq 0$ polynomial equalities/inequalities and express g as an algebraic system of the form

$$\begin{cases} \big[P_*(x_0,x_1,\ldots) =, \neq 0\big] \text{or} \ldots \text{or} \big[P_*(x_0,x_1,\ldots) =, \neq 0\big], \\ \quad \ldots \quad \ldots \\ \big[P_*(x_0,x_1,\ldots) =, \neq 0\big] \text{or} \ldots \text{or} \big[P_*(x_0,x_1,\ldots) =, \neq 0\big], \end{cases} \tag{1}$$

where P_* denote arbitrary polynomials in the free variables $x_0, x_1, x_2, \ldots$ associated with the $GF(2^k)$ inputs of the circuit.

The algebraic constraints on the inputs are also given as CNF, and form an algebraic system of the same type.

g is constant'0' if and only if
{input constraints CNF} AND {g-subgraph CNF} (2)
is unsatisfiable.
g is constant'1' if and only if
{input constraints CNF} AND NOT{g-subgraph CNF} (3)
is unsatisfiable.
Each of the expressions (2) and (3) can be converted to a single CNF of the form (1). Hence, it suffices to show how to check whether a system of the form (1) is unsatisfiable.

Satisfiability checking algorithm:

The first step is to get rid of inequalities in the system. For each inequality $P_*(x_0, x_1, \ldots) \neq 0$ we introduce an auxiliary free variable t_* and replace the inequality by

$$t_* \cdot P_*(x_0, x_1, \ldots) - 1 = 0.$$

One can easily check that if the system before replacement is satisfiable in variables $\{x_0, x_1, \ldots, t_0, t_1, \ldots\}$ then the system after replacement is satisfiable in variables $\{x_0, x_1, \ldots, t_0, t_1, \ldots\} \cup \{t_*\}$ and vice versa.

The new system contains only polynomial equalities. Next we replace all OR operations with multiplication:

$$\begin{cases} \big(Q_*(\{x_*\},\{t_*\})\big) \cdot \ldots \cdot \big(Q_*(\{x_*\},\{t_*\})\big) = 0, \\ \quad \ldots \quad \ldots \\ \big(Q_*(\{x_*\},\{t_*\})\big) \cdot \ldots \cdot \big(Q_*(\{x_*\},\{t_*\})\big) = 0, \end{cases}$$

Now we have a regular algebraic system of multivariate polynomials over $GF(2^k)$.

By Hilbert's Weak Nullstellensatz a system of multivariate polynomials is unsatisfiable over an algebraically closed field if and only if the ideal generated by the polynomials of the system coincides with the whole ring (i.e. contains 1) (refer [8], pp. 169-173), $x \in GF(2^k)$ if and only if $\big[x \in \mathrm{alg\_closure}\big(GF(2^k)\big) \text{ AND } x^{2^k} - x = 0\big]$. For each variable v_* of our system add equation $v_*^{2^k} - v_* = 0$. The new system (denote it S) is satisfiable in the algebraic closure of $GF(2^k)$ if and only if the original system is satisfiable in $GF(2^k)$.

Next we build a Gröbner basis of the ideal given by the polynomials of system S. This can be done by Buchberger's algorithm ([8], pp. 77, 82-87). The original system is unsatisfiable in $GF(2^k)$ if and only if the Gröbner basis of S contains 1.

If the value of g is proved to be a constant '0' or '1' assign this value to g (computation successful). Otherwise fork on the unresolved Boolean signal g as follows:

Add two copies of the current check at the end of the check file as given below.

- If g is an input Boolean signal add `g <= '0'` to the "In bits settings" section of copy_1 and `g <= '1'` to the "In bits settings" section of copy_2.
- Otherwise add NOT(System (1)) to the conjunctive normal form in "Algebraic constraints on GF inputs" section of copy_1 and System (1) to the CNF in "Algebraic constraints on GF inputs" section of copy_2.

Skip the current check and continue to the next one with the two additional checks added at the end of the queue. As a side note, the two examples in subsections III-A and III-B do not require branching of this type for completion.

The only operation we have not explained yet is division. **DIV** : In logical circuits division is usually implemented as `if` $y \neq 0$ `return x/y; else return 0;` (which is equivalent to xy^{2^k-2}). To compute the result of division we first attempt to prove that the constraints on the inputs imply that the divisor is either always $= 0$ or always $\neq 0$ by the same algebraic method as for the gates with Boolean output. If successful, we simply assign 0 or the rational $expression x/y$ to g. Otherwise we fork on the test of [denominator $= 0$] the same way as shown above for non-input Boolean signals.

We have shown how to compute value of any gate given the values of its inputs. $GF(2^k)$ signals are computed as symbolic rational expressions in the input signals, and Boolean signals must compute to constant '0' or '1' creating new branches with additional algebraic constraints on the inputs if necessary. This completes the description of our algorithm.

Our actual C implementation contains many more features than described above. The most important ones include:

- Careful manipulations of conjunctive normal form systems: A brute force manipulation of CNFs, and opening parenthesis in polynomial products which come from large OR-clauses would cause an immediate exponential explosion of the size of the system. However special care is taken of systems of the form (1) which most commonly appear in algebraic circuits. This prevents a rapid increase of the size of the system - at least for typical cases. In particular, if g-CNF has only one OR clause of length $\geq$ 2, i.e. has the form $\Big([P_* =, \neq 0]\,\text{or}\ldots\text{or}\,[P_* =, \neq 0]\Big)$ and $[P_* =, \neq 0]\,\text{and}\ldots\text{and}\,[P_* =, \neq 0]$, our implementation ensures the size of any system for which we build a Gröbner basis is simply equal to the sum of the sizes of the input constraints system and g-CNF system.
- "Lazy" signal computation method: In order to find values of expressions such as ('1' $or\ x$), ('0' $and\ x$), (when '1' : $const$ else x) etc., we do not compute x. This gives a significant speed up especially when the signals whose values we need to verify are localized in a relatively small part of a large circuit.
- Verification flow control: The user can control a number of verification process options such as whether to spend more time on Gröbner basis computation of a given bit vs. fork; whether to attempt to save time by skipping the $x \in GF(2^k)$ constraints which makes false negatives (but not false positives) possible; etc.

The verification process can have three possible outcomes:

1) For all checks all crucial bit values are computed and match the expected values.
2) One of the checks (including checks added by "fork on unresolved bit") fails because the value of one of the crucial bits is opposite to the expected value specified in the check file.
3) One of the checks (including checks added by "fork on unresolved bit") fails to compute one of the crucial bit values due to insufficient time or memory.

In the latter two cases an interactive bug tracing interface allows the user to browse the graph of signals and view their values in the form of symbolic rational expressions and algebraic systems.

III. Experimental Results

If there is no restriction on time and memory the verification process is guaranteed to prove or disprove the specification in the check file. We will give in what follows two simple examples (subsections III-A and III-B) where this is accomplished within a reasonable amount of time, demonstrating the power of reasoning at the Galois field level as opposed to the Boolean level. For complex, real-life designs (as exemplified in subsection III-C) we have found it useful to help BLUEVERI by manually partitioning the search space, resulting in very little use of the "forking" feature described earlier. In addition, in some instances care is taken to specify the circuit in otherwise equivalent forms to aid BLUEVERI in keeping down the size of its internal rational expressions and the complexity of algebraic systems it generates; this was not necessary in the two examples below.

A. The uncorrectable error flag of a sample Reed-Solomon decoder

As a first example, we consider a Reed-Solomon code with symbols belonging to a finite field $GF(q)$ with $q = 2^k$ elements for some integer k. We shall assume that the length of this code is $n = 2^k - 1$. Let r denote the number of check symbols of the Reed-Solomon code. We assume that this Reed-Solomon code has been furnished with a decoder that is capable of correcting any one symbol error, and can detect up to $r-1$ different errors. This decoder has a number of different components, one of which is responsible for the computation of the uncorrectable error flag. This flag is a single Boolean output that is raised whenever the decoder has detected 2,3, or up to $r-1$ errors, and kept low whenever the error scenario corresponds to a single error, or alternately whenever there is no error.

For our choice of Reed-Solomon code, the r syndromes of this Reed Solomon code can be computed from a (potentially corrupted) encoded vector $v \in GF(q)^n$ using the formula

$$S_i = \sum_{j=0}^{n-1} v_j \omega^{ij}$$

for $i \in \{0, \cdots, r-1\}$, where ω denotes a primitive element of the field. Furthermore, letting $e \in F_q^n$ denote the *error vector* affecting v, so that $v = e + x$ where $x \in F_q^n$ is the uncorrupted codeword it is also known that due to linearity of the addition operator in finite fields and the vector that x has zero syndrome,

$$S_i = \sum_{j=0}^{n-1} e_j \omega^{ij} \tag{4}$$

The design of the uncorrectable error flag for this scenario is a well understood problem; for the sake of demonstration we deduce what might be a reasonable method to test it directly through formal methods. It can be easily seen from (4) that if there is only one error in e then the syndromes satisfy the following condition: $S_i S_{i+2} = S_{i+1}^2$ for $i = 0, \cdots, r-3$. Furthermore it is also known whenever e has at least one error and at most r errors, one or more of the $\{S_i\}_{i=0}^{r-1}$ is nonzero. This leads to the conjecture that one can compute the uncorrectable error flag through the following code, written using BLUEVERI VHDL style semantics:

```
t_comp : for i in 0 to r-3 generate
  t(i) <= add(mult(s(i),s(i+2)),square(s(i)));
end t_comp;
snz <= is_nz(s(0)) or ... or is_nz(s(r-1));
tnz <= is_nz(t(0)) or ... or is_nz(t(r-3));
UE  <= snz and tnz;
```

As written above, `snz` and `tnz` represent two distinct systems of equations which BLUEVERI will treat independently of each other. On the other hand, BLUEVERI will attempt to establish whether `tnz` (for example) is true or false by examining the properties of `t(0) ... t(r-3)` simultaneously as opposed to testing whether each `t(i)` is zero or not individually.

In order to test the ability of a model checker to prove the correctness of this implementation of the uncorrectable error flag, we assume that the syndrome generation portion of the decoder has been proved correct separately; this task is in fact generally computationally simpler than the one currently at hand. We then build a module that accepts inputs `e_m(0)...e_m(t-1)` (for the error magnitudes) and inputs `l(0)...l(t-1)` (for the error locations) where t is the maximum number of errors one can inject into the decoder during the test; in this particular example for the uncorrectable error flag to be correct it is known that $t = r-1$. This module emulates the syndrome generator and computes `s(0)...s(r-1)` using the equation `s(i)` $= \sum_{i=0}^{t-1}$ `l(i) e_m(i)` (as per Equation 4), and then passes the resulting syndromes to a module that computes the uncorrectable error flag as previously described.

In order to test a variety of error scenarios, we can place constraints on `e_m(i)` and `l(i)`. For example, one can restrict the test to have exactly two errors by specifying the following constraints:

```
e(0) != 0, e(1) != 0, l(0) != 0, l(1) != 0
add(l(0),l(1)) != 0, e(1) = ... = e(t-1) = 0
```

Note that in a field of characteristic 2, addition is equivalent to subtraction, and hence the addition constraint effectively constrains `l(0) != l(1)`. These constraints can be specified in a BLUEVERI check file as equal/not equal to zero conditions on multivariate polynomial expressions. When BLUEVERI examines the dependencies of the `UE` signal, it finds that it depends on `snz` and `tnz`. BLUEVERI must either resolve that both are true, or that at least one of them is false. As described earlier, this is accomplished by invoking an attempt to compute the Gröbner basis of various system of equations related to the constraints and the expressions defining `snz` and `tnz`. Similar experiments can be conducted by updating the constraints to specify "at least two, but not more than y errors" where y is a number between 2 and $r-1$.

In order to test the capability of BLUEVERI as applied to this problem and contrast it with that of a formal prover (we chose SixthSense, IBM's state-of-the-art formal and semi-formal verification tool set, for that purpose), we set up a test with $r = 8$, $b = 8$ and with the capability to inject from 2 up to 7 errors at arbitrary locations, since the corresponding Reed-Solomon decoder is supposed to be able to detect all those errors. We also set up a parallel test with $b = 4$ which is a considerably simpler problem for a Boolean oriented formal verification system such as SixthSense [11]. The SixthSense and BLUEVERI experiments do not have any special tuning of the VHDL or the tool to improve the outcomes.

We refer the reader to Table I where the experiments were performed in a single processor (POWER6 processor @ 5GHz running AIX) and the SixthSense was run as a single software thread mainly orchestrating redundancy removal and SAT algorithms. In this set of experiments, BLUEVERI was configured to reason about the circuit with the variables (due to inputs or constraints) belonging to the *algebraic closure* of the fields. This in essence means that we did not constrain the variables to belong to the field $GF(256)$ (resp. $GF(16)$) depending on whether the symbols used were 8 bit (resp. 4 bit) symbols. The consequence of this is that although the BLUEVERI results are listed under 8-bit column, they in fact hold *for any field size*, including larger field sizes which would be even harder for a bit-level verification system to handle. Both formal systems were able to prove the correctness of the uncorrectable error flag under the single error scenario quite easily, but SixthSense was not able to prove the correctness of this flag in the double error case in the amount of time indicated in the table. In order to test the sensitivity of SXS to the field size, we performed a similar experiment for a Reed-Solomon code defined over $GF(16)$. In this case we saw better results from SixthSense, since we were able to prove the correctness of double and triple error detect cases but not four error case. It is worth noting that the field size determines many important properties of an error control code, including

symbol errors	expected UE	8 bit symbols			4 bit symbols	
		BLUEVERI	input bits	SXS	input bits	SXS
1	false	Success after 0.1 s.	16	Success after 14 s.	8	Success after 0.7 s
2	true	Success after 1 s.	32	Gives up after 24 h.	16	Success after 3 s
3	true	Success after 1 s.	48	N/A	24	Success after 55 m
4	true	Success after 33 m.	64	N/A	32	Gives up after 24h
5	true	Gives up after 6 h.	80	N/A	40	N/A

TABLE I

EXPERIMENTAL RESULTS FOR THE FORMAL VERIFICATION OF THE UNCORRECTABLE ERROR FLAG OF A SINGLE ERROR CORRECT, MULTIPLE ERROR DETECT REED-SOLOMON DECODER. SXS REFERS TO SIXTH SENSE, A BIT-LEVEL FORMAL VERIFICATION TOOL SET DEVELOPED AT IBM.

errors	8 bit symbols			4 bit symbols	
	BLUEVERI	input bits	SXS	input bits	SXS
2	Succ. 2 s.	32	Gives up after 24h	16	Succ. 0.6s
3	Succ. 2.1 s.	48	N/A	24	Succ. 16m
4	Succ. 2.1 s.	64	N/A	32	Gives up after 24h
5	Succ. 2.3 s.	80	N/A	40	N/A
6	Succ. 3.1 s.	96	N/A	48	N/A
7	Succ. 49.4 s.	112	N/A	56	N/A
8	Succ. 8m	128	N/A	64	N/A
9	Succ. 53m	144	N/A	72	N/A

TABLE II

EXPERIMENTAL RESULTS FOR THE FORMAL VERIFICATION OF THE ERROR MAGNITUDE COMPUTATION STAGE OF A REED-SOLOMON CODE.

the total codeword length, and thus it cannot be modified for the purposes of formal verification since the resulting code is entirely different and, in all likelihood, not applicable to the original problem.

B. Computing error magnitudes in a Reed-Solomon code

One of the tasks that an error control decoder for a code defined over multibit ($q > 2$) symbols must perform is to compute the locations of the symbols in error and then to compute the multibit pattern that one must add to those locations in order to correct the codeword. This multibit pattern is called the *error magnitude*. Suppose that there are t errors in a codeword, and let `s(0)`$,\cdots,$`s(t-1)` be the first t syndromes (note that this example is for a different setting than the example in the previous subsection). From (4), we can derive that error magnitude computation can be carried over using the equation

$$\begin{bmatrix} \texttt{e\_m(0)} \\ \vdots \\ \texttt{e\_m(t-1)} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \texttt{l(0)} & \cdots & \texttt{l(t-1)} \\ \vdots & \ddots & \vdots \\ \texttt{l(0)}^{t-1} & \cdots & \texttt{l(t-1)}^{t-1} \end{bmatrix}^{-1} \begin{bmatrix} \texttt{s(0)} \\ \vdots \\ \texttt{s(t-1)} \end{bmatrix}$$

The inverse matrix above can be derived analytically. It is well known that the inverse is non singular if and only if the locations `l(i)` are all distinct of each other. This restriction can be specified through $\binom{t}{2}$ constraints each of which is a polynomial with two monomials. We refer the reader to Table III-B where we show that in this case, BLUEVERI was able to show the correctness of the corresponding circuit with up to 8 errors, while SixthSense was unable to finish the double error case within the time allocated. As in the previous subsection, in this particular example the result for BLUEVERI is actually field size independent since it exploits only the algebraic properties of the symbols. It is worth noting that the Gröbner basis machinery in BLUEVERI does get involved in proving the correctness of this circuit. This is because the inversion of the Vandermonde matrix results in rational expressions (as opposed to plain polynomial expressions) whose denominator could be zero. The task of Gröbner in here then is to show that the denominator is not zero given the assumptions on the inputs, so that BLUEVERI can proceed with the corresponding algebraic simplifications leading to the desired result.

C. A note on a real life application of BLUEVERI

The examples in the previous subsections are meant to illustrate the capabilities of a formal verification system such as BLUEVERI when compared to Boolean oriented systems. In our experience, the implementation of a real-life encoder/decoder employs many custom algorithm variants as one tries to address problems that are specific to the application at hand. In the most significant application of BLUEVERI so far, we have succeeded in proving the correctness of an ECC of a POWER microprocessor that is based on the mathematics of Reed-Solomon codes. The correctness criteria included all correctable and uncorrectable cases for which we had given guaranteed behavior (e.g. recovery from complete chip failures and detection of multiple errors). The ECC, from the decoders perspective, had over 1000 bits of input including several tens of bits worth of configuration parameters. The number of syndrome bits produced by the decoder was over 100 bits, although our testing did include testing the behavior of the encoder with analytically generated symbolic syndromes, it was not limited to it - approximately half of the total testing time exercised the more than 1000 bits of input of the circuit directly. The number of Galois field and Boolean elements in the corresponding graph is over 100,000 (compared to at most a few hundred in the previous experiments). Because of the complexity of the problem, we had to case-split to create 1M different tests, each of which exercised formally a particular region of the test space. It took about 2 weeks to prove the correctness of the entire design in a 10 machine Linux (x86) cluster.

IV. TECHNICAL SOLUTIONS

The BLUEVERI tool leverages IBM's existing front-end and simulation tools and flows. For language processing we are using Portals, IBM's HDL compiler, which accepts the synthesizable subset of standard VHDL and Verilog languages. Portals performs behavioral synthesis on procedural HDL and produces an elaborated netlist, for BLUEVERI this is in the DADB logic database. DADB is a box-pin-net logic database used for verification flows, such as topology checking and simulator model build, which supports client transforms via a dynamically loaded plugin architecture.

Portals was modified for BLUEVERI to support the black-boxing of function calls, enabling the logic to be represented in a form amenable to analysis by BLUEVERI. High level language constructs which are output by Portals into the netlist, such as case statements, can be synthesized into lower level representations by the use of DADB client transforms.

The BLUEVERI analysis tool has its own custom input netlist format. A netlist translator was built as a DADB client to enable the tools flow from Portals into BLUEVERI.

The MESA logic simulator is a high performance cycle simulator used for functional verification within IBM. MESA simulation models are built from logic netlists in DADB by using model build clients.

The BLUEVERI code is written in C. For the computation of Gröbner bases we use ”SINGULAR” [10] a powerful program for algebraic geometry computations distributed under general public license. BLUEVERI runs SINGULAR as a child process and uses ”EXPECT.h”, (a standard C library), for sending queries and receiving results from SINGULAR’s Gröbner basis engine. The Gröbner basis obtained is for the inverse degree lexicographical ordering.

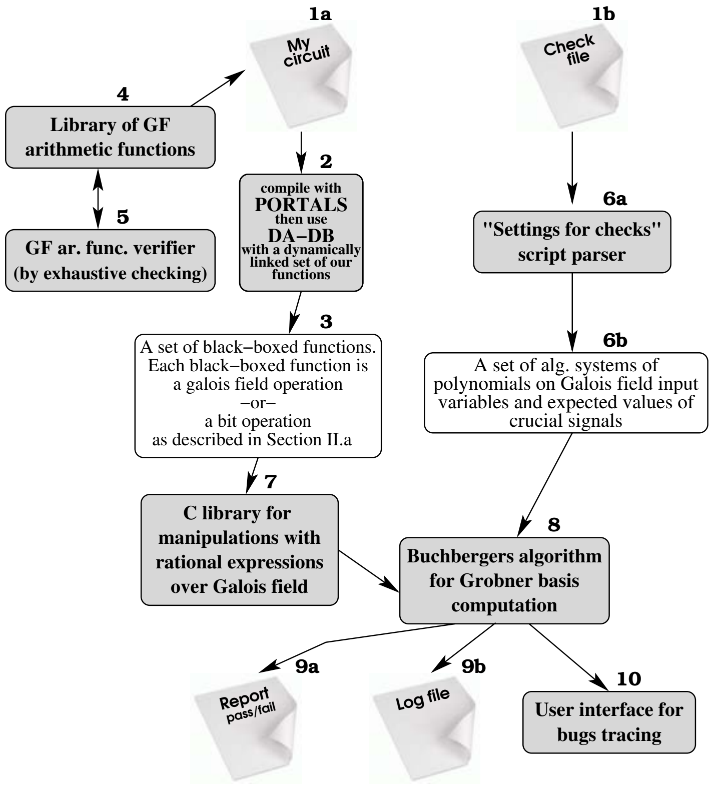

1a
My circuit
1b
Check file
4
Library of GF arithmetic functions
2
compile with PORTALS then use DA–DB with a dynamically linked set of our functions
5
GF ar. func. verifier (by exhaustive checking)
6a
"Settings for checks" script parser
3
A set of black–boxed functions. Each black–boxed function is a galois field operation –or– a bit operation as described in Section II.a
6b
A set of alg. systems of polynomials on Galois field input variables and expected values of crucial signals
7
C library for manipulations with rational expressions over Galois field
8
Buchbergers algorithm for Grobner basis computation
9a
Report pass/fail
9b
Log file
10
User interface for bugs tracing

Fig. 3. General schema of BLUEVERI tool.

V. Conclusions

In this article we presented a novel technique for designing and verifying circuits based on the mathematics of Galois fields. At the heart of our approach is the idea of exposing operations on Galois field directly to a verification layer (encapsulated in a tool called BLUEVERI) which leverages powerful techniques from algebraic geometry to reason about the properties of the abstract Galois field rational expressions generated in the circuit. Our circuits are specified using a subset of existing Hardware Description Languages and as such, remain fully synthesizable, an important attribute to reduce the possibility of human error in the design process.

We demonstrated the value of the ideas we proposed in the context of two problems representative of the type of situations encountered when designing error correcting codes. In both instances, we showed BLUEVERI can significantly outperform conventional bit-level formal verification. We outlined a successful application of the BLUEVERI system to prove correctness of a real production complex error correcting code implemented on a POWER microprocessor which otherwise could not be verified conclusively with traditional verification methods.

VI. Acknowledgments

The authors would like to thank Barry M Trager, Shmuel Winograd and Geert Janseen of IBM Research for insightful discussions to help shape the solution.

References

[1] Meaney, P. J. and Lastras-Montaño, L. A. and Papazova, V. K. and Stephens, E. and Johnson, J. S. and Alves, L. C. and O’Connor, J. A. and Clarke, W. J., *IBM zEnterprise redundant array of independent memory subsystem* IBM Journal of Research and Development, Jan-Feb, Vol. 56, 2012.

[2] Lastras-Montaño, L.A.; Meaney, P.J.; Stephens, E.; Trager, B.M.; O’Connor, J.; Alves, L.C., *A new class of array codes for memory storage*, Information Theory and Applications Workshop (ITA), 2011 , vol., no., pp.1-10, 6-11 Feb. 2011

[3] R. E. Bryant *Graph Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on Computers, vol. C-35, pp. 677691, August 1986.

[4] R. E. Bryant and Y-A. Chen *Verification of Arithmetic Functions with Binary Moment Diagrams*, Design Automation Conference 1995.

[5] U. Kebschull and W. Rosentiel *Efficient graph-based computation and manipulation of functional decision diagrams*, European Conference on Design Automation, pp. 278 - 282, 1993.

[6] S. Morioka, Y. Katayama and T. Yamane *Towards Efficient Verification of Arithmetic Algorithms over Galois Fields*. Proc. Computer Aided Verification 2001, vol. 2102, pp.465-477.

[7] Jinpeng Lv, Priyank Kalla and Florian Enescu *Verification of Composite Galois Field Multipliers over $GF(2^{m^n})$ Using Computer Algebra Techniques*. Proc. IEEE International High Level Design Validation and Test Workshop 2011, pp.136-143.

[8] David Cox, John Little, and Donald O’Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer, 2010. ISBN: 0-387-35650-9.

[9] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Encyclopedia of Mathematics and Its Applications, Volume 20. Cambridge University Press, 1997. ISBN: 0-521-39231-4.

[10] Decker, W.; Greuel, G.-M.; Pfister, G.; Schönemann, H.: SINGULAR 3-1-3 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de (2011).

[11] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, *Scalable automated verification via expert-system guided transformations*, Formal Methods in Computer-Aided Design, 2004, pp. 159173.

Symbolic Trajectory Evaluation: The Primary Validation Vehicle for Next Generation Intel® Processor Graphics FPU

M Achutha KiranKumar V, Aarti Gupta, and Rajnish Ghughal
Intel Corporation
{achutha.kirankumar.v.m, aarti.gupta, rajnish.ghughal}@intel.com

Abstract— **Formal Verification (FV) is widely acknowledged for improving validation effectiveness. Usually formal verification has been used to supplement more traditional coverage oriented testing activities. Arithmetic Data-path FV has matured over the time to completely replace traditional dynamic validation methodologies. Moreover, it gives an additional promise of 100% data-space coverage. Symbolic Trajectory Evaluation (STE) is the best proven method of FV on Intel® data-path designs. The Floating Point Units (FPUs) are generally very data-path intensive. In the next generation Intel Processor Graphics design, the FPU was completely re-architected and this necessitated a methodology which could guarantee complete verification in a tight verification schedule. STE was brought in to meet this formidable target. This paper discusses the efficient application of this methodology to achieve convincing results. More than 201 bugs were caught in a very short verification cycle using STE.**

I. INTRODUCTION

EVER since Intel graphics moved from chipset to CPU, there is an ever-increasing demand on the graphics design to make the combination of CPU and graphics more compelling for the end user. The current generation graphics processor unit (GPU) is not just solely used for image rendering but also to share the workload with core-CPU processor [1, 2]. Graphics processor designs have very short design cycles to cope with the market requirements. In this paper, we address the problem of verifying large arithmetic data-path circuits using formal verification techniques in such short design cycles.

Intel microprocessor design cycles follow a uniform methodology over successive generations, known as “tick-tock cadence” [3]. In a typical “tock” part of this cadence, major innovative architectural changes are introduced in the microprocessor design. In a typical “tick” part of this cadence, relatively less architectural changes are introduced while design is moved to the next generation semi-conductor manufacturing process technology. This cadence effectively allows consistently improving next generation microprocessor capabilities and performance.

The latest “tick” CPU processor of Intel encases a graphics engine that can be called “tock” taking into account the number of architectural changes that went into the design. Such aggressive architectural changes were introduced to provide significantly increased graphics performance. This presented a huge challenge to the verification team to verify these architectural changes in a relatively shorter time. STE-based formal verification methodology was used to tackle this challenge providing a high degree of confidence in the correctness of this design.

Execution units performing arithmetic computation inside graphics microprocessors are becoming more available to end users for high-performance computing using general purpose graphics processor unit (GPGPU) programming methodology. This makes it much more critical to ensure that the next generation Intel Processor Graphics design implements the arithmetic standards faithfully and the stakes are much higher than previous generation graphics designs if a really tricky bug were to be missed in the graphics execution unit [4, 5, 6].

This paper talks about how this challenging task of validating “tock” features in “tick” timeline, was simplified and successfully accomplished by making use of STE. We describe how STE was used to establish correctness of floating-point data-path circuits which resulted in discovery of 201 bugs. Many of these bugs were truly “FV-quality” bugs which would have never been found by other forms of validation or discovered much later in the project cycle. Similar bugs were discovered very late in the post-silicon phase in previous generation graphics design where STE-based formal verification was not applied. Discovery of these bugs in the latest graphics design has greatly contributed to achieving higher RTL quality way ahead of tape-out[1] and significantly reducing the risk of encountering them in the post-silicon verification.

A. Related Work

STE based formal verification approach has been widely used at Intel in the past for various microprocessor designs to formally verify data-path designs [7, 8, 9, 10, 11, 12]. It has been proven very effective at handling large arithmetic circuits and establishing their correctness against a formal specification and discovering very difficult to find bugs in the

[1] Sending design for semi-conductor manufacturing production is referred to as tape-out.

process which would have been undetected by any other form of validation. For example, STE-based formal verification was used in an execution cluster of Intel microarchitecture code named Nehalem to replace traditional simulation [7].

At Intel, FV techniques have also been applied to formally verify designs other than arithmetic data-path in microprocessor using other forms of formal verification, e.g., pipeline scheduler verification, cache coherence protocol verification etc. [19, 20, 21]. These formal techniques typically involve using explicit state model-checking, symbolic model checking or bounded model checking using SAT. In our experience, these techniques are not as suitable as STE for verifying industrial scale floating point arithmetic data-path designs.

Formal verification of floating-point arithmetic designs is a well-studied problem both at Intel and elsewhere in the industry [7, 8, 9, 10, 23, 24, 25, 26, 27] due to the critical need of correctness of floating-point arithmetic. Majority of these work [7, 8, 9, 10, 24, 25, 26] concentrate on verifying floating-point addition, multiplier or divider operation but do not address floating-point fused multiply addition operation which presents a lot of unique challenges of its own.

In [23], formal verification of FMA operation is done by excluding multiplier from the cone of influence and hence the proof of the correctness of multiplication is missing. In our experience, proof of the correctness of multiplication, especially for double precision floating-point arithmetic is a very challenging task and is critical to verify. In [23], a key assumption was to disallow other operations in the pipeline before or after the FMA operation. Our work allows arbitrary operations to come before and after the FMA operation in pipeline. In fact, some of the most interesting bugs that we found involved interaction between FMA and other operations in the pipeline. Such bugs are near impossible to discover by any other forms of validation and hence it is critical that such limiting assumptions should not be employed in formal verification of floating-point arithmetic designs. One of the many such bugs discovered by our work is described in a later sub-section of this paper (see Complex Interaction Bugs).

In [27], Slobodova describes a FMA formal verification proof developed at Intel previously using STE. This approach mirrors closely with the approach used by us with some key differences. FMA design implementation described in [27] was significantly simpler than the FMA design in the next generation Intel graphics, which uses an approach known as "sea of multipliers" to implement very power-efficient and latency-optimized multiplication. Such a FMA design challenged us to approach the problem of verifying booth-encoded partial products generation completely differently than similar efforts in the past. Also in [27], FMA operation on denormal floating-point numbers was not formally verified due to limited hardware support of denormal floating-point numbers in the design under consideration. In the next generation of Intel graphics design, FMA operation fully supports denormal floating-point numbers in the hardware. This significantly expanded the data-space of the problem and required us to completely rethink the traditional case-split strategy employed in floating-point addition operation from ground-up. In addition, a lot more floating-point precisions are supported in the next generation Intel graphics design than the design under consideration in [27].

Despite STE's success in formally verifying arithmetic designs in microprocessors previously, its application to graphics design projects has been limited. This paper presents first such application to large-scale industrial graphics design where formal verification was used as a primary method of validation resulting in a very large number of high quality bugs found in the process.

II. What is STE?

Symbolic Trajectory Evaluation (STE) is a formal verification method originally developed by Seger & Bryant in 1995 [13]. It is a high-performance model checking technique using a symbolic simulation-based approach [14, 15, 16]. It works over binary decision diagrams (BDDs), which are symbolic Boolean expressions. STE is particularly well suited to handle data-path properties, and it is used to verify gate-level models against more abstract reference models.

A. *Technical Framework*

Formal Verification of data-paths in the design under test (DUT) is done using the Forte framework, originally built on top of the Voss system [14]. The framework and methods built around it are depicted in Figure 1.

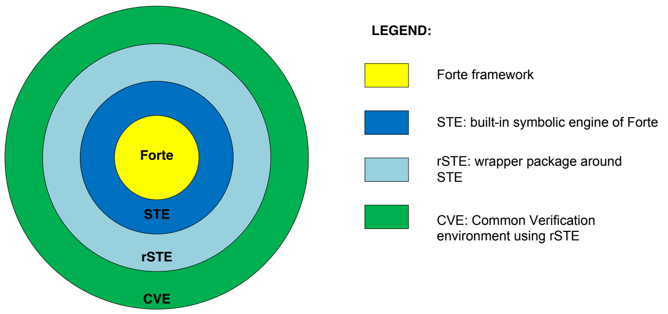

Forte
STE
rSTE
CVE
LEGEND:
Forte framework
STE: built-in symbolic engine of Forte
rSTE: wrapper package around STE
CVE: Common Verification environment using rSTE

Figure 1 Building Blocks of STE Infrastructure

The interface language to the Forte is reFLect (FL for short), a lazy, strongly-typed functional language in the ML family [18]. The Forte framework directly supports symbolic simulation on circuit models through STE as a built-in function.

Relational STE (rSTE) is a package built around STE to support relational specifications. Effectively, rSTE is a tool to check whether a set of constraints ("the input constraints"), implies another set of constraints ("the output constraints") over all traces of the circuit. It provides sophisticated debug support, breakpoints etc. It also provides a number of capabilities to manage the complexity of the formal verification tasks.

The Common Verification Environment (CVE) was developed to create a standard, uniform methodology for writing specifications and carrying out verification tasks using STE. The CVE is built upon a generic abstract model of the DUT (design under test). The CVE combines proof engineering and software engineering to create a standard, uniform methodology for writing specifications and carrying out verification tasks. The aim of the effort is to support reuse and code maintenance over a constantly changing design, and separate common and project-specific parts to allow shared code to be written only once. The CVE collects all verification code to a single common directory structure and provides a platform to share code across projects.

B. *Verification flow using STE*

The basic flow-diagram of verification using STE is shown in Figure 2. STE checks that given a set of constraints, the symbolic simulation output of the DUT matches the given specifications or not. Constraints define the behavior of input nodes (src_nodes) at arbitrary input time (src_time). For a particular data-path to be tested, nodes that may take variable values are driven symbolic values, nodes those are required to be fixed are driven constants(0/1), and all other nodes that don't fall in cone of influence are made don't care (X). Specifications express requirements that should hold on output nodes (wb_nodes) at writeback time (wb_time = src_time + latency of data-path). The set of constraints are applied to the specification which are spec constraints. Constraints and specifications are written by the user in FL. STE computes a symbolic representation for each node (n,t), extracts node-time information at writeback (wb_ckt) and checks against the writeback specification (wb_spec) provided by the user. The result could be either a full proof or a counter example or X as depicted in the Fig.2. The X signifies one of the three options: (1) Circuit output results in X which is undesirable or (2) the antecedent needs refinement or (3) the BDD size just blew out of proportions of the defined weakening limit and hence complexity reduction techniques had to be employed to get it under control.

It is quite often that the verification engineer needs to prove properties of the intermediate states of the data-path design in order to be able to prove correctness of the final result. These properties are written as *invariants* and proven using either inductive methods using STE or as a data-path property. Discovery and proofs of these invariants play a key role in enabling formal verification of data-path designs.

STE has been extremely successful in verifying properties of circuits containing large data-paths. FPU validation using STE in the next generation Intel Processor Graphics design produced exceptional and unprecedented results. Section IV describes the story of this success and path taken to achieve it.

III. Next Generation Graphics Processor FPU

The bulk of processing in a graphics processor is done by an array of programmable cores or Execution Units (EUs). The main processing engine of an EU is its Floating Point Unit (FPU). FPU performs the desired operation by means of executing the micro-instructions (uops) launched by the EU. The goal of FPU validation is to verify the results of these uops.

A. *Graphics Processor FPU Validation Challenges*

The FPUs of the graphics processor are data-path intensive and getting complete vector coverage on all the operations is almost impossible, even with multibillion-cycle dynamic simulation runs. In addition to this, with the introduction of Compute Shaders (CS), more stringent precision requirements are now imposed on FPUs to comply with various standards like IEEE standard for binary floating-point arithmetic, Open Computing Language (OpenCL®), Open Graphics Library (OpenGL®), DX11, etc. [4, 5, 6]. Before the introduction of Compute Shaders, the FPU operations were limited to executing instructions for the 3D. But now, the FPUs are

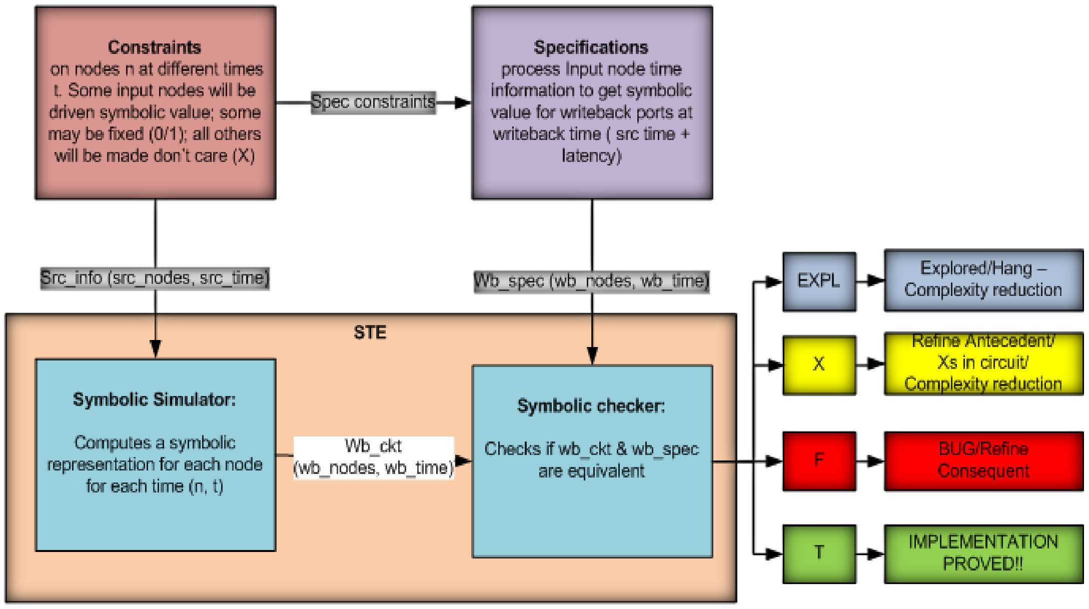

Constraints
on nodes n at different times t. Some input nodes will be driven symbolic value; some may be fixed (0/1); all others will be made don't care (X)
Spec constraints
Specifications
process Input node time information to get symbolic value for writeback ports at writeback time (src time + latency)
Src_info (src_nodes, src_time)
Wb_spec (wb_nodes, wb_time)
STE
Symbolic Simulator:
Computes a symbolic representation for each node for each time (n, t)
Wb_ckt (wb_nodes, wb_time)
Symbolic checker:
Checks if wb_ckt & wb_spec are equivalent
EXPL
Explored/Hang – Complexity reduction
X
Refine Antecedent/ Xs in circuit/ Complexity reduction
F
BUG/Refine Consequent
T
IMPLEMENTATION PROVED!!

Figure 2 Basic flow diagram of verification methodology using STE

exposed to general purpose applications similar to the CPU cores and the accuracy/precision requirements have become more exacting. The challenge in validating the FPU data-path is to get 100% coverage while meeting the precision/accuracy requirements.

Though the CVE provides a common base and methodology for implementing uops, the implementations vary from project to project and design-specific intricacies had to be taken care of. The graphics instruction set[2] is compact but has a complex format. The instruction format had a number of qualifiers which were not present in a CPU instruction. Challenges faced due to these additional qualifiers for the implementation of the GT STE are explained in the Table1 below.

Table 1: GT Specific challenges for STE deployment

GT intricacy	Brief description
Support for Various Dsizes	Unequal Dsizes for sources/destinations
Flag Generation/ interpretation	In addition to IEEE flags, GT also supports flag output based on outputs
Source modifiers	Negation, absolute, negation of absolute
Saturation	Floating point saturation allowed for GT
Accumulator Source	Allows implicit/explicit accumulator source
Accumulator Destination	Allows implicit/explicit accumulator destination
Denorm Handling	Non uniform for different precisions
ALT mode	Support for non-IEEE compliant mode
NaN Handling	Fixed NaN output for some operations
Rounding modes	Instruction specific rounding
Channel enables	Selective enabling of FPU pipelines

Apart from the above common validation challenges of any graphics processor validation, the next generation Intel graphics processor faced a new set of validation changes due to huge architectural changes done for better graphics performance. Performance improvement of graphics directly translates to enhancing the raw execution power source of the graphics engine i.e. EU. FPU which is the main data-cruncher of EU was completely re-designed for the next generation Intel Processor Graphics design to get the desired performance improvement and area-reduction per EU. This overhaul of design and architecture imposed a lot of validation challenges. Some of the major design change categories in FPU are described in Table 2.

Due to the complete redesigning of FPU in latest GPU design, validation was considered as a high risk to be completed with high confidence level. Data-path formal verification using STE was brought in to the rescue.

Table 2: FPU Specific changes in next generation Intel Processor Graphics design

FPU Changes	Validation Risk
FPU Pipeline Restructure	**High**
Increased Conformance to Arithmetic Standards	**High**
Improved Programming capability	**Medium**
Improved Clock Gating	**Low**
Area, Power & Throughput optimizations	**Low**

IV. Operation FV Bug-Hunt

The following section explains how STE enabled an early validation of the design and how it helped in unearthing a wide variety of bugs. The methodology was applied on the design where it passed the basic check-in gates and ready for mass regression. STE proof regressions were run on every released model and the failures were debugged.

A. Proof readiness before the design and validation reference models

Like any other design methodology, the new graphics design followed a phased implementation of new design features (DCNs). Thus register transfer level (RTL) hardware design was under constant churn and so was the C++ based golden reference model for dynamic validation[3] (DV). Because of the following remarkable qualities of STE, we were ready to develop proofs before RTL or DV Reference Model was ready:

1. **One proof – many projects:**
   The beauty of CVE is the specification code reusability across projects. The specification of processor micro-operations doesn't change much over the generations of design. As most of the proofs are agnostic to the implementation details, they are easily portable to any project with/without minor changes. Many graphics-specific integer and floating-point (FP) STE proofs were developed during the previous generation Intel Processor Graphics verification timeframe. Most of these proofs could be seamlessly integrated into the new graphics design verification with minor modifications. Though we were not ready with full set of proofs, we were equipped enough to do the basic checking and getting RTL to a stable state.

2. **One proof-wider coverage:**
   Just like any other Formal methodology, STE doesn't depend on any scalar vectors for simulation. It takes

[2] Graphics instruction set is for internal consumption and not exposed for external reference.

[3] Dynamic validation refers to the traditional method of doing verification using simulation over concrete (as opposed to symbolic) input values.

into account all the control signals and results in a comprehensive coverage. Just one proof can provide the control space coverage for all signals in the cone of influence of the operation being checked, in addition to the comprehensive data space checking. During the first month of verification cycle, we focused on developing and regressing formal proofs to check correctness for simple operations (For example, logical operations like OR, AND, integer add, etc.), to get the RTL healthy. This simple operation checking itself unearthed much more number of bugs in different areas of the design as compared to the dynamic validation which was being run in parallel on the full instruction set of FPU. Once the basic proofs started passing, we embarked on proving the formal proofs for more complex operations (like floating point conversions to integer/floats, floating point add, mul, mad, etc.). Regressions were run on every new model and the failures were debugged and reported out. A passing proof guarantees 100% coverage of the input data space within the defined constraints of control logic.

3. **Capability to mask unimplemented features:**
During Front End Development all the new design features are implemented in a phase-wise manner. Validation needs to be in close tandem with the design implementation to verify only the implemented features. STE provides the user with the capability of selectively masking the unimplemented features through addition of simple constraints. This enabled us to make uninterrupted forward progress in validation. Once the proofs are passing, the constraints are phased out as the RTL matured with the planned implementation.

4. **Ease of debugging:**
The counter examples provided by the tool were very intuitive and could easily help in reproducing the failure in dynamic simulation. The in-house developed AGM viewer utility aids in debugging through waveforms and schematics and was of great help in debugging.

B. STE as monster bug-hunter

STE could help in stabilizing the RTL quality by regressing over every design iteration and point out the failures in different areas. A wide range of bugs varying in both quantity and quality were unearthed in the process. The bugs ranged from bugs on controls related to data-path, instruction interaction bugs, clock gating bugs to deep corner case scenarios. Some of these bugs are mentioned below to highlight the uniqueness of the bugs found:

1. **Clock-gating Bugs:**
The new graphics design implements very aggressive clock gating and bugs were found on logic with flops gated with incorrect pipeline signals, unintended gating and non-uniform gating across the data.

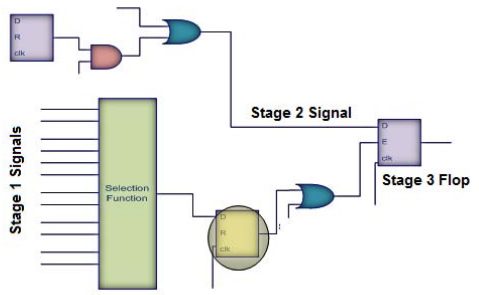


Figure 3: Clock Gating Bug Example

As an example, in the scenario depicted in Figure 3, the buggy RTL missed the flop shown in the highlighted circle. While the data input of stage 3 flop received a stage 2 signal, the signal that drives the enable input was of stage 1. Dynamic simulation couldn't catch this miss, as all the flops were initialized to zero during reset phase. As STE simulation would work with symbols driven at the inputs, the resultant of the above logic would result in X[4]s at the flop output. Reproduction of the similar scenario in dynamic simulation wasn't a straight forward task.

2. **Data space Corner Cases:**
Majority of the bugs found using STE are deep corner case scenarios. Finding deep-rooted data space issues is one of the most sought after features of STE.
To mention one example, a particular evasive bug in a three source floating point operation "OP (A, B, C)" manifested itself only when the following data requirements were met:

```
A = 0x1cc9_9398_0003_3273
B = 0x1ff4_04b2_5a15_c2bb
C = 0x8000_0000_0000_0001
```

The probability of hitting this specific data requirement is 1 in 2^{192} ($2^{64}*2^{64}*2^{64}$) possibilities. The chance of reaching this kind of scenario with any other validation methodology is very remote.

3. **Complex Interaction Bugs:**
This category of bugs manifest when two operations occur one after another with specific data requirements on the sources for each of these operations. Due to the nature of the source supplied to each of these operations, a certain incorrect behavior in the design is exposed that would only manifest when these two operations are in close temporal proximity to each other.
One such specific interaction bug was found when a particular two source operation "OP1 (A, B)" produced incorrect results, when it was immediately preceded by

[4] X is introduced by STE to automatically abstract symbolic computation that may not be relevant for the verification task.

a particular three source operation “OP2 (C,D,E)” and the input data of both these uops followed the data requirements given below:

A, B, C, D, E are floating-point numbers below.

Conditions on Preceding Operation:

- Operation must be OP2 (C, D, E)
- C is negative
- C is not Infinity/Not a Number (NAN)/Zero

Conditions on Current Operation:

- Operation must be OP1 (A, B)
- A or B is a negative NAN

OP2 must come in the cycle immediately before OP1

This was a rare combination of “Instruction Interaction” and “Data space Corner-case” issue. Such scenario with specific data requirements on current and previous operations is almost impossible to be caught by any other validation methodology.

4. **Initialization Bugs:**
This set of bugs relates to erroneous initialization of state elements in the design. One example of these kinds of bugs is explained in Figure 4. The figure illustrates priority selection logic where a raw move (a move operation without any modifiers or qualifiers) has a higher precedence to create a data valid (dv) signal. The integer to float conversion signal was missing in this cone of logic of the buggy RTL. Usually, the dynamic tests start with initializing the configuration registers which are usually raw move instructions and hence the flop in this logic would get initialized and the int2float conversion in these tests would run as expected. On the contrary, the STE simulation signal would see Xs on the dv signal, oblivious to the preceding instructions.

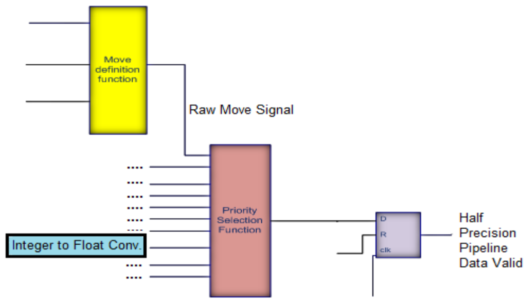


Figure 4: Initialization Bug Example

5. **Control Logic Bugs:**
This set of bugs is the result of faulty control logic in the circuit. The usual sources of these bugs are typos in the RTL or incorrect bug fixes.

These bugs are not hard to detect by other validation methodologies as they don’t have very stringent data requirements and can be reproduced by just appropriate setting of control parameters. But still some of these bugs evade capture by other methodologies because of their random nature.

STE, however, guarantees complete coverage of data and control variables and makes sure that these bugs are weeded out. These kinds of bugs are usually found in the first formal verification attempt for the concerned operation.

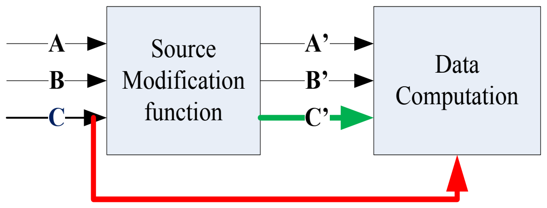


Figure 5: Control Logic Bug Example

One simple example of such bug is presented in Figure 5. In this case, due to a typo mistake in the RTL, one of the sources was taken for data computation without applying a source modification function which was the design requirement.

V. Results

The results achieved by applying STE early in the design cycle are explained in the sections below:

A. Comparison against contemporary methodologies

In addition to STE, FPU validation in graphics projects is carried out by a set of other standard validation methodologies. Table 3 gives a short summary of these techniques.

Table 3: List of Contemporary Validation techniques for graphics FPU validation

Validation Technique	Methodology	Reference Model
DV1	Dynamic stress validation using targeted vectors generated by Intel Internal Tool	DV C++ based Reference Model + Intel Internal Floating Point Library
DV2	Dynamic coverage-based validation using controlled random vector generation by Intel Internal Tool	DV C++ based Reference Model

DV3	Dynamic validation using standard random test bench features of System Verilog	DV C++ based Reference Model
FV1	Another Formal Verification Approach with C++ specification against the RTL	C++ based specification

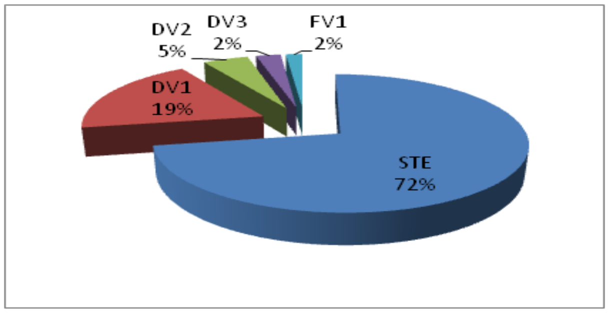


Figure 6: RTL bugs found by validation methodologies

Figure 6 gives the distribution of RTL bugs exposed and filed by different methodologies for pre-silicon verification in the new graphics processor.

Of the total number of discrepancies found, STE takes the lion share with 72% of the bugs being exposed by this methodology. The bugs which were found by the other methodologies were from:

1. Operations which were not verified by STE.
2. A very small set of RTL bugs in the areas covered by STE, were found because either the STE proof was under development or they were debugged ahead of STE failures.

As we approached the end of the project cycle, we reviewed all the constraints with the designers and refined them. These could also catch a good deal of issues in the design. We are yet to implement an automated way of converting the constraints to SVA based monitors.

As evident from the Figure 6, STE formed the backbone of major feature validation for FPU. Almost 3 out of the 4 RTL bugs filed in the new-GPU FPU were found by STE. The confidence on STE verifying uops were so high that the rest of the methodologies were realigned to target only those areas which were not covered through STE.

STE was the tool of choice from the RTL side for any optimizations in the micro-architecture. Any optimizations for timing fixes, and power optimizations were run first through STE and based on our feedback, the fixes were either selected or rejected for functionality. STE helped in maintaining the health of the RTL and could avoid the downtrends which are typically seen in any of the design projects.

B. *Bug Distribution*

Figure 7 depicts the division of 201 bugs found by STE in the next generation Intel Processor Graphics FPU. Though majority of them were RTL bugs, we also found ample issues with the Spec (the architectural specification) and the golden DV Reference Model.

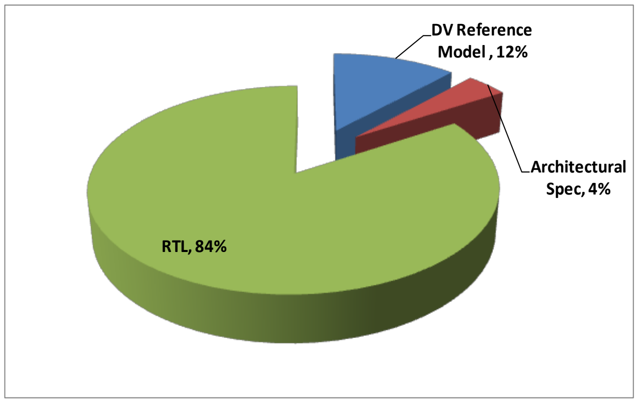


Figure 7: Distribution of 201 STE bugs

There are a decent number of bugs filed on DV Reference Model. These bugs were found through STE when bugs caught by STE were tried to be reproduced on dynamic simulation. Dynamic simulation runs tests on both RTL and Reference Model and any difference in the results between these two models are reported as error. If the DV Reference Model implementation also bears the bug, then Reference Model would be in unison with RTL and the bug would be masked. The bugs that are filed from STE on DV Reference Model fall in a category which exposed the issues where the Reference Model also has the bug like the RTL and the masked issue would never get exposed in any of the other methodology. Hence, we could cleanse not only the RTL but also the golden model which is used by other methodologies too.

Architectural Specification bugs were found by STE when the defined pseudo code of an operation in Spec didn't match with the standard CVE proofs. Since, CVE proofs conform to most of the arithmetic standards and have been verified in variety of projects, some of the failures turned out to be Spec issues. The whole execution was carried out by a two member team during a span of 9months and the man-year effort is comparable and even lesser than what has been observed in STE validation on EU in CPU projects. Thanks to the reusability of the CVE.

C. *Forward and Backward Compatibility of Proofs*

Once the proofs were completely developed, we could execute them on some of the earlier projects which were currently under post silicon debug and found some issues. The proofs developed are broadly compatible with generations of graphics designs, both forward and backward.

VI. Summary

The next generation Intel processor graphics FPU was completely redesigned to comply with arithmetic standards, increase programmability and to optimize on latency, power and area. This paper detailed the architectural complexities introduced due to the design improvements. A comprehensive mechanism was needed to validate this new design in a short time span. This paper discussed how STE was used as a primary validation vehicle on FPU to thwart out issues in the RTL and specifications by early deployment in the project cycle. More than 200 bugs were unearthed by STE in this project. To date, we haven't found any bug escape in the uops verified by STE nor any spec bugs found through DV, which boosts our confidence in the tool and its capabilities to achieve zero post silicon bugs.

Our experiences through the project execution confirm the fact that if STE is implemented early in the project design cycle, it could stabilize the RTL earlier. A reusable proof that is ready before the RTL and validation environment helps in early bug hunting and improving the quality early in the project, which means significant improvement in the effectiveness of validation. We strongly believe that the effectiveness of STE for improved quality of validation would prove valuable in the validation of a wide range of designs. We proved that it is possible to reach a better level of quality with a lower investment of resources, thereby reducing the overall cost of validation.

Acknowledgment

Sincere thanks to Tom Schubert and Roope Kaivola for training us on STE and providing constant support. We would like to acknowledge Naveen Matam, Ashutosh Garg, Jin Xu and Shubh Shah for helping us with micro-architecture and design front. We would like to thank Durairaghavan Kasturirangan and Archana Vijaykumar for giving us the opportunity to perform this work.

References

[1] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU Computing: Graphics Processing Unit--powerful, programmable, and highly parallel—are increasingly targeting general-purpose computing applications", *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008.

[2] Martin Rumpf and Robert Strzodka, "Graphics Processor Units: New Prospects for Parallel Computing", *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, Springer-Verlag, 2005.

[3] Intel Tick Tock Model, [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html .

[4] *IEEE standard for binary floating-point arithmetic*, ANSI/IEEE Std 754-1985, 1985.

[5] OpenCL - The open standard for parallel programming of heterogeneous systems, [Online]. Available: http://www.khronos.org/opencl/

[6] OpenGL, [Online]. Available: http://www.opengl.org/

[7] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodov´a, Christopher Taylor, Vladimir Frolov, Erik Reeber and Armaghan Naik, "Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation", *CAV '09 Proceedings of the 21st International Conference on Computer Aided Verification*, Sept 2009.

[8] R. Kaivola, "Formal verification of Pentium® 4 components with symbolic simulation and inductive invariants" *In CAV, Computer Aided Verification*, volume 3576 of *LNCS*, pages 170–184. Springer, 2005.

[9] R. Kaivola and M. D. Aagaard. "Divider circuit verification with model checking and theorem proving" In *TPHOLs*, volume 1869 of *LNCS*, pages 338–355. Springer, 2000.

[10] R. Kaivola and K. Kohatsu, "Proof engineering in the large: formal verification of Pentium® 4 floating-point divider", *Int'l J. on Software Tools for Technology Transfer,* 4:323–334, 2003.

[11] R. Kaivola and A. Naik, "Formal verification of high-level conformance with symbolic simulation", In *HLDVT, IEEE International Workshop on High-Level Design Validation and Test*, pages 153–159, 2005.

[12] R. Kaivola and N. Narasimhan, "Formal verification of the Pentium® 4 floating-point multiplier", In *DATE, Design, Automation and Test in Europe*, pages 20–27, 2002.

[13] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories", *Formal Methods in System Design*, 6(2), 1995.

[14] S. Hazelhurst and C.-J. H. Seger, "Symbolic trajectory evaluation," in *Formal Hardware Verification*, T. Kropf, Ed. New York: Springer Verlag, 1997, ch. 1, pp. 3–78.

[15] J. O Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. (1999, First quarter). "Formally verifying IEEE Compliance of floating-point hardware", Santa Clara, CA: Intel Corp. [Online]. Available: http://developer.intel.com/technology/itj/

[16] R. B. Jones, "Symbolic Simulation Methods for Industrial Formal Verification", Kluwer Academic Publishers, 2002.

[17] Carl- Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme, "An Industrially Effective Environment for Formal Hardware Verification", *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, Sep2005, Vol. 24 Issue 9, p1381-1405, 15p.

[18] L. Paulson, *ML for the Working Programmer,* Cambridge University Press, 1996.

[19] Robert Beers, Rajnish Ghughal and Mark Aagaard. *Applications of Hierarchical Verification in Model Checking,* Formal Methods in Computer-Aided Design 2000. Lecture Notes in Computer Science, 2000, Volume 1954/2000, 1-19, DOI: 10.1007/3-540-40922-X_1

[20] Robert Beers, *"Pre-RTL formal verification: An Intel experience,"* Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE , vol., no., pp.806-811, 8-13 June 2008

[21] B. Bingham, J. Bingham, F. de Paula, J. Erickson, M. Reitblatt, and G. Singh, ``*Industrial Strength Distributed Explicit State Model Checking*'', International Workshop on Parallel and Distributed Methods in Verification (PDMC) 2010.

[22] Slobodova, A.; Davis, J.; Swords, S.; Hunt, W.; , *"A flexible formal verification framework for industrial scale validation,"* Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on , vol., no., pp.89-97, 11-13 July 2011

[23] Jacobi, C.; Weber, K.; Paruthi, V.; Baumgartner, J.; , *"Automatic formal verification of fused-multiply-add FPUs,"* Design, Automation and Test in Europe, 2005. Proceedings , vol., no., pp. 1298- 1303 Vol. 2, 7-11 March 2005

[24] Warren A. Hunt, Sol Swords, Jared Davis and Anna Slobodova: *"Use of Formal Verification at Centaur Technology".* Design and Verification of Microprocessor Systems for High-Assurance Applications. 2010, 65-88, DOI: 10.1007/978-1-4419-1539-9_3

[25] Russinoff, D. M., Hunt, W. A., & Johnson, S. D. *A case study in formal verification of register-transfer logic with ACL2: the floating point adder of the AMD AthlonTM processor.* Formal Methods in Computer-Aided Design. Third International Conference, FMCAD 2000. Proceedings.

[26] D. Russinoff, M. Kaufmann, E. Smith, and R. Sumners. *Formal verification of floating-point RTL at AMD using the ACL2 theorem prover.* In Nikolai Simonov, editor, Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, July 2005.

[27] Anna Slobodova. *Challenges for Formal Verification in Industrial Setting.* Formal Methods: Applications and Technology Lecture Notes in Computer Science, 2007, Volume 4346/2007, 1-22, DOI: 10.1007/978-3-540-70952-7_1

Enhanced Reachability Analysis via Automated Dynamic Netlist-Based Hint Generation

Jiazhao Xu Mark Williams Hari Mony Jason Baumgartner
IBM Systems & Technology Group

Abstract— **While SAT-based algorithms have largely displaced BDD-based verification techniques due to their typically higher scalability, there are classes of problems for which BDD-based reachability analysis is the only existing method for an automated solution. Nonetheless, reachability engines require a high degree of tuning to perform well on challenging benchmarks. In addition to clever partitioning and scheduling techniques, the use of *hints* has been proposed to decompose an otherwise breadth-first fixedpoint computation into a series of underapproximate computations, requiring a larger number of (pre-)image iterations though often significantly reducing peak BDD size and thus resource requirements. In this paper, we introduce a novel approach to boost the scalability of reachability computation: automated netlist-based hint generation. Experiments confirm that this approach can yield significant resource reductions; often over an order of magnitude on complex problems compared to reachability analysis without hints, and even compared to SAT-based proof techniques.**

I. Introduction

Since the advent of symbolic model checking more than two decades ago, automated verification tools have evolved dramatically in capacity. This evolution is due to a variety of innovations, including (in extreme brevity) advanced BDD-based techniques [1], [2], SAT-based proof [3], [4] and falsification engines [5], [6], [7], a variety of simplification and abstraction techniques to reduce problem complexity [8], [9], [10], and a modular transformation-based tool architecture to allow all of the above to synergistically decompose a complex verification problem [11] under guidance of advanced orchestration techniques [12]. Clever software engineering techniques, parallel processing, and more powerful computers upon which to run these tools have also played an important role. This boost in *scalability* has yielded a boost in *usability*, proliferating model checking from a craft requiring dedicated verification expertise to pervasive use even by non-experts, e.g., for lighter-weight assertion-based verification or sequential equivalence checking. Even state-of-the-art academic solvers such as ABC [13] and PdTrav [14] have become quite powerful through the above techniques.

The advent of unbounded SAT-based proof techniques such as interpolation [3] and IC3 [4] has played a particularly pronounced role in the scalability of contemporary model checkers. Whereas BDD-based reachability analysis tends to become impractical if the design under verification cannot be reduced or abstracted below several hundred state variables, SAT-based techniques on occasion can scale beyond tens of thousands of state variables. Nonetheless, BDDs may dramatically outperform SAT-based techniques for classes of problems, thus a well-tuned BDD-based reachability engine is an essential component of a state-of-the-art verification tool.

Numerous techniques have been developed to boost the scalability of BDD-based reachability engines. Examples include using a partitioned transition relation (TR) instead of a monolithic representation [1], advanced quantification and conjunction scheduling based upon metrics such as variable dependency [2], as well as heuristics to balance splitting and conjoining strategies [7]. The application of BDD-reduction operators such as *bdd_constrain*, *bdd_restrict* and *bdd_compact* [15] on the transition relation and state set representations have also yielded substantial scalability improvements.

The concept of *hints* was presented in [16] as a method to mitigate the BDD size explosion that often happens during intermediate steps of breadth-first reachability analysis, despite the BDDs being much more compact at early and even late stages. The intuition behind this phenomenon is that breadth-first analysis explores many disjoint design behaviors in parallel, causing asymmetries and thus bloat in the intermediate BDD representations – whereas the final reached state representation may have many asymmetries “filled in” hence be more compact. Hints are used to iteratively constrain the transition relation and thereby *direct* the symbolic search, computing states reachable along the constrained transition relation from those reached using prior hints. Completeness is ensured by finally restoring the original transition relation once the hints have been exhausted. Despite requiring a greater number of (pre-)image computations, this compaction of intermediate BDDs results in fewer and less-expensive dynamic variable ordering computations. These benefits collectively often reduce resources for complex problems, in cases enabling a solution for problems which would otherwise exhaust time or memory limitations. As noted in [16], as concurs with our practical experience: even *arbitrary* hints often reduce complexity for difficult problems. This preliminary work proposed the use of manually-generated hints based upon design insight.

This work was extended toward automation in [17], [18]. In [17] the authors propose to analyze the control-data flow graph of a behavioral Verilog design, using *branch conditions* as hints that effectively decompose the design similar to program slicing techniques. [18] extends this approach by using these conditions as a *known-complete* disjunctive partitioning, without requiring the final fixedpoint computation where the unconstrained transition relation is used to ensure completeness. While demonstrated as effective on a set of designs, these approaches are of limited practical applicability since they require a high-level behavioral design format which may not be available. This becomes prohibitive in application domains such as sequential equivalence checking which may require analysis of post-synthesis netlists, and within a transformation-based verification toolset which may have applied numerous reduction and abstraction techniques to aggressively shrink the original netlist to something feasible for BDD-based analysis. These approaches also may not be suitable for classes of designs which are harder to program slice such as those with highly-pipelined or multi-threaded behavior.

Other techniques have also been proposed to reduce peak BDD size through departing from breadth-first search, such as high-density reachability analysis [19]. This technique resorts to intermediate under-approximate reachability analysis, partitioning images when BDD sizes exceed a threshold. Our practical experience with such approaches is that they suffer convergence problems (e.g., requiring a virtually-unbounded number of image computations) rendering them of limited practical utility. In contrast, a benefit of hints is that their impact on the number of image computations may provably be linearly bounded given proper controls.

In this paper, we introduce a novel automated dynamic hint generation approach to boost the scalability of reachability computation. In contrast to [17], [18], our work is focused upon generating high-quality hints from *arbitrary* netlist representations, and is triggered on-demand only when reachability computation exceeds a resource threshold. We have used this technique successfully both for property checking and sequential equivalence checking. Our specific contributions, as detailed in Section III, include a method to dynamically introduce hints to the reachability process based upon resource thresholds; dynamic algorithms to compute effective hint sequences from a transition relation; and a method to truncate reachability analysis under a given hint if it is deemed to risk increasing the number of overall image computations by too large a factor. While these techniques are all heuristic in their attempt to reduce the complexity of a reachability computation, our experiments in Section IV confirm that they often significantly boost performance for complex problems, and in many cases outperform SAT-based techniques.

II. Preliminaries

A model checking problem may be expressed as a *netlist*: a directed graph whose nodes (termed *gates*) comprise primary *inputs*, *state elements*, and a variety of combinational logic operators. State elements have associated *initial values* and *next-state functions*. A *state* is a Boolean valuation to the state elements. An *initial state* is a state consistent with the conjunction of the initial values.

The *transition relation* $TR(x, i, y)$ associated with a netlist comprises *current state variables* $\{x_1, \dots, x_m\}$; *next state variables* $\{y_1, \dots, y_m\}$; and *input variables* $\{i_1, \dots, i_n\}$. It is defined in a straight-forward way from the next-state functions of the state elements of the netlist.

An *image computation* is used to compute the successors of a set of states s, defined by $\exists i. \exists x. TR(x, i, y) \wedge s$. A *reachability computation* may be performed by first setting the partial set of reached states to the initial states, then growing that set by iteratively computing its image to add to the partial set via union.

Algorithm 1 Reachability using Hints

```
 1: function FORWARDREACH(TR, hints, init_states)
 2:     reached = init_states

 3:     // true will be the last-used hint in hints
 4:     while (hint = pop(hints)) do
 5:         hint_TR = apply_hint(TR, hint) // constrain TR with hint
 6:         frontier = reached // first image with hint_TR uses reached

 7:         while (true) do
 8:             image = compute_image(hint_TR, frontier)
 9:             frontier = compute_frontier(image, reached)

10:             if (frontier is empty) then break
11:             end if

12:             reached = bdd_or(reached, frontier)
13:         end while
14:     end while
15: end function
```

III. Enhanced Reachability Algorithms

In this section we present our automated hint generation algorithms. Algorithm 1 depicts a traditional framework for reachability analysis using hints [16]. In a traditional application, the hints are manually provided to the reachability process, and the final hint must be *true* (or *constant 1*) to ensure that the original transition relation will be *restored* for a complete reachability computation.

There are several limitations of the use of hints in practice which we address in this paper.

1. Requiring manual specification of hints diminishes their utility, and enabling automation only for problems of suitable Verilog syntax [17] is limiting in practice. We thus introduce in Section III-A an effective automated hint generation algorithm which operates directly upon the transition relation, and in Section III-B an algorithm which iterates through the generated hints.

2. In cases, hints may degrade performance of the reachability computation because they increase the number of image computations, while not significantly reducing effort vs. unconstrained image computation. For easier problems, this is a risk because the image computations are already efficient. For complex problems, a fixed set of hints may not adequately simplify image computation, whereas a more aggressive set of hints may be helpful. To address this issue, we introduce in Section III-C a reachability framework which introduces hints upon demand, when BDDs exceed configurable thresholds.

3. In rare cases, hints may result in convergence problems for reachability computation. A pathological example is for a counter with a *parallel load* port, where any arbitrary state may be loaded into the counter under control of a particular input – otherwise it may take an exponential number of steps to transition from one reachable state of the counter to another. If a hint disables that parallel load, it may dramatically increase the number of necessary image computations for a fixedpoint

Algorithm 2 Hint Introduction Algorithm

```
    function GENERATE_HINTS(TR, hints, reached, reduction_limit, var_limit)
2:      vars = all variables that are not in hints

        for all BDD variable var in vars do
4:          compute rank of positive and negative literals of var
            add best rank literal to array ranks
6:      end for

        sort ranks // ranks used to generate the initial hint cube
8:      hint_cube = bdd_1 $\wedge \bigwedge$ hints // form cube for already-selected hints

        while (|ranks|) do
10:         literal = best candidate from rank
            prune rank from ranks
12:         new_cube = hint_cube $\wedge$ literal

            if (new_cube contradicts TR or reached) then
14:             compute rank for opposite literal polarity; add to ranks
                re-sort ranks; continue

16:         else
                hint_cube = new_cube
18:             hints = hints $\cup$ literal
                compute TR size reduction of TR from new_cube

20:             if ((TR reduction exceeds reduction_limit) or (|hints| exceeds var_limit)) then
                    break
22:             end if
            end if
24:     end while

        return hints
26: end function
```

computation, slowing overall progress. We thus introduce in Section III-D a mechanism to truncate the use of a specific hint prior to fixedpoint if necessary, for overall robustness.

A. Automated Hint Generation Algorithm

Algorithm 2 outlines our automated hint-generation technique. The hints that we have found most effective are *BDD cubes* over input variables and/or current state variables. A BDD cube is a conjunction of BDD literals (positive or negative) over a set of BDD variables.

There are several heuristics that we have found effective for selecting the best BDD literals to include in hints. One heuristic is to first select a BDD variable using the *use_count* of that variable as its ranking measure; i.e., the number of BDD nodes associated with a given variable, then to select the positive or negative literal of that variable based on the amount of reduction to the transition relation each literal provided. The intuition of using this metric is that it provides an indication that asymmetries over the corresponding variable may be the cause of intermediate BDD growth. Another heuristic is to rank all the BDD literals according to the criteria of how much reduction a given variable cofactoring provides to the transition relation, which in turn provides an estimate of how much they may speed up image computation. We have empirically found that the former works best. The ranking metrics, along with the most promising variable polarity, are recorded in the *ranks* data structure against which each BDD variable will be sorted.

After ranking the BDD variables the next task is to select a set of BDD literals to form the first hint cube. This is not merely a matter of choosing the k highest-ranked literals from the sorted *ranks* data structure, as the result may yield a "contradicting" hint cube which has an empty intersection with the transition relation or *reached* set, which begins as the initial states. We thus perform a consistencycheck on the candidate *hint_cube* before adding a literal to it, and in case of a contradiction, we flip the polarity of that variable and re-rank. To avoid adding more literals to the first hint cube than necessary, we use two termination criteria: **(1)** *reduction_limit* measures the degree to which the given *hint_cube* reduces the *TR*, i.e. $\big(sizeof(TR) - sizeof(TR \wedge hint\_cube)\big)/sizeof(TR)$, and **(2)** *var_limit* which provides an upper-bound on the number of literals to be added to *hints*. Our experience shows that *reduction_limit* may be left large, on the order of 100% since a small transition relation will be fast for reachability computation anyway, and a *var_limit* of between 10 and 15 literals yields the best results for larger netlists (see further discussion in Section IV and Figure 3). We use the *bdd_and* operation to constrain the transition relation with the hint cube. Since the hint is merely a cube, the *bdd_and* operation is as effective as other BDD constraining operations.

It is noteworthy that the generated hints are highly dependent upon BDD variable ordering. This algorithm may be called multiple times in the overall reachability framework as per Algorithm 4, possibly adding additional hints to a non-empty set of previously-generated hints. It is likely that dynamic variable ordering was invoked between these calls, hence the added hints will reflect the best choice under the current ordering.

B. Hint Iteration Algorithm

In addition to deciding the set of literals that will be used for the hints, it is important to decide the *sequence* of hints that will be applied given this set. We have found the most consistently-effective hint-successor strategy to be iteratively eliminating literals from the original hint cube, thus starting the computation with a maximally-restrictive constraint and gradually relaxing that constraint. This observation was formed over years of relying upon the use of manual hints for BDD-based reachability analysis in practice, prior to the availability of more scalable alternative proof techniques. It is consistent with the intuitive notion that hints should be introduced to decompose an overly-complex fixedpoint computation from following many disparate design behaviors to focusing on a smaller yet growing set of behaviors. This process is depicted in Algorithm 3.

The *gen_next_hint* function is called as part of the overall reachability framework in Algorithm 4. When reachability analysis exceeds a complexity threshold and hint literals are generated, this algorithm determines the sequence in which literals are removed from that set for successive hints. Note also that the sequence of applied hints is *dynamically* determined, vs. merely deciding a fixed order when hint literals are generated via Algorithm 2, as variable ordering may have changed between those points.

Algorithm 3 Hint Successor Algorithm

```
    function GEN_NEXT_HINT(TR, hints, first, reached)
        cur_hint = bdd_1 $\wedge \bigwedge hints$

3:      if (first) then
            return cur_hint
        end if

6:      re-rank and re-sort hints

        while (|hints|) do
            remove lowest-rank literal from hints
9:          next_hint = bdd_1 $\wedge \bigwedge hints$

            if ((next_hint $\wedge$ reached) $\subseteq$ (cur_hint $\wedge$ reached)) then
                continue // next_hint is vacuous

12:         else
                return next_hint
            end if
15:     end while

        return bdd_1
    end function
```

In our experiments, we observed occasional occurrences of "*vacuous*" hints which do not add any new states to the reached set. Rather than waste resources performing a useless image and frontier computation in such cases, we developed an inexpensive test to detect most vacuous hints and avoid generating them. This test consists of computing the conjunction I_c of *cur_hint* with the reached set, and checking if I_c contains the conjunction I_n of the candidate *next_hint* with the reached set. If so, *next_hint* is vacuous and we proceed to the next literal. Since our hints are cubes, this computation is efficient in practice. Empirically, we found that approximately 20% of candidate hints are vacuous, and this step results in approximately 15% improvement in overall performance.

C. Dynamic Hint Introduction

In practice, a *monolithic* application of hints may not be ideal for several reasons. First, for easier problems, the use of hints often degrades performance of the reachability computation because they increase the number of image computations, while not significantly reducing effort compared to the unconstrained image computations. In other cases, the application of hints is inadequate to reduce the complexity and make image computation tractable. We thus have developed a framework which introduces hints only upon demand, as BDD sizes exceed configurable thresholds.

We exploit a "node limit" feature provided by our BDD package which limits the peak number of nodes it is allowed to generate within a BDD operation. If an operation exceeds this limit, a special *UNKNOWN* handle is returned, which is treated similarly to the X value in ternary analysis. Every image computation is performed using a node limit, which allows that computation to add at most a fixed number of BDD nodes. If the image computation returns *UNKNOWN*, additional hint literals are generated to mitigate the BDD explosion, and the constrained image computation is repeated. Our practical experience is that the threshold should not be too small, nor overly large; an allowance of 350000 nodes is the best setting we have practically found. To allow convergence on very complex problems, whenever we generate hints, we increase this threshold by a configurable factor (50% is effective) to avoid future hints from being triggered too frequently on problems that intrinsically need large BDDs. This process is depicted in our overall reachability flow in Algorithm 4, under control of variable *bdd_threshold*.

D. Hint Truncation

In a traditional hint application as per Algorithm 1, a full fixedpoint of states reachable under the corresponding hint-constrained transition relation is performed for each hint. However, in cases, a hint may dramatically increase the number of necessary image computations as per the example of a counter with *parallel load* capability discussed in Section III. For robustness, we thus have found it useful to place a limit on the maximum number of image computations that are allowed for a given hint. Because it is difficult to predict the number of image computations which would be necessary without hints (i.e., the *diameter* of the design), this metric in practice can be kept quite large (on the order of 10000), and optionally increased every time this limit is encountered. Using such a facility, one may thus ensure that the use of hints increases the number of image computations vs. reachability without hints by at most a linear factor. This process is depicted in our overall reachability flow in Algorithm 4, under control of variable *hint_iters*.

E. Overall Enhanced Reachability Algorithm

Algorithm 4 summarizes our overall enhanced reachability framework, combining aspects described in prior sections. Compared to Algorithm 1, the primary differences are: **(1)** automated generation of hints (Section III-A); **(2)** dynamically-prioritized iteration among the generated hints, taking into account current variable ordering (Section III-B); **(3)** dynamic triggering of hint introduction (Section III-C); and **(4)** truncation of a hint if too many image computations are required under that hint (Section III-D).

IV. Experimental Results

In this section we provide experimental results to illustrate the effectiveness of our techniques. These experiments are all derived from the Hardware Model Checking Competition 2011 benchmarks [20], pruned to the 92 that: **(1)** were not trivially solved by light-weight logic optimizations or random simulation; **(2)** could complete a reachability computation either with or without hints within a 4 hour time limit and 4GB memory: and **(3)** our dynamic hint-generation algorithm was invoked due to resource requirements. Because these benchmarks are provided in AIGER form, not in behavioral Verilog syntax, the technique of [17] is not applicable. Furthermore, the benchmarks used in [17] are a small set that are not publicly available, hence we could not readily contrast our approaches. restrict our focus to those of [20].

We implemented our techniques in the reachability engine included in the IBM verification tool *SixthSense* [12]. This

Algorithm 4 **Reachability using Dynamic Automated Hints**

```
1: function FORWARDREACH(TR, init_states, var_limit, bdd_threshold,
   bdd_growth_factor, depth_threshold, reduction_limit, hint_iters)
2:     reached = init_states
3:     hints = emptyset
4:     first = true

5:     while (hint = GET_NEXT_HINT(TR, hints, first)) do
6:         first = false
7:         hint_iters = 0
8:         hint_TR = apply_hint(TR, hint) // constrain TR with hint
9:         frontier = reached

10:        while (true) do
11:            image = compute_image(hint_TR, frontier, bdd_threshold)

12:            if (image ≡ UNKNOWN) then // aborted due to bdd_threshold
13:                bdd_threshold = bdd_threshold * bdd_growth_factor
14:                hints = GENERATE_HINTS(TR, hints, reached, reduction_limit, var_limit)
15:                first = true
16:                break
17:            end if

18:            hint_iters++

19:            if (hint_iters ≥ depth_threshold) then break // goto next hint
20:            end if

21:            frontier = compute_frontier(image, reached)

22:            if (frontier is empty) then
23:                if (hints ≡ emptyset) then return // fixedpoint complete
24:                end if
25:                break // goto next hint
26:            end if

27:            reached = bdd_or(reached, frontier)
28:        end while
29:    end while
30: end function
```

engine uses an internally-developed BDD package [21], with standard features such as dynamic variable ordering, as well as more advanced techniques such as support for multiple distinct BDD "managers" with the ability to cast BDDs from one to the other as long as they share the same set of variables, though not necessarily in the same order. One occasion to use a different BDD manager is for on-the-fly counterexample generation when concurrently solving multiple properties, to avoid trace generation from triggering a dynamic variable ordering which hurts continued reachability analysis. An initial ordering of the variables is computed using the interleaved approach described in [22]. We use the transition relation partitioning techniques of [2] by default. Cutpointing is supported in both the transition relation and the BDD representing the property.

A number of optimizations are used during the reachability computation to reduce BDD size, including backward and forward pruning of the transition relation as described in [23]. In addition, we make use of the BDD reduction operations described in [15] when computing frontiers.

Figures 1 and 2 summarize our experiments for runtime and memory, respectively, of performing a reachability computation after light-weight logic optimization techniques, with and without our dynamic hint generation approach. Note that we forced a complete reachability computation on each of these, even if an on-the-fly failure could have enabled early termination. None of these experiments exhausted memory, though there were timeouts which are omitted from Figure 2. These results demonstrate that hints do introduce a computational overhead for simpler problems – primarily those which complete within several minutes. However, for a majority of the complex problems, hints significantly improve runtime and memory requirements. In fact, the benefit achieved by hints is largely proportional to the complexity of the verification problems: those which would otherwise require approximately 1000 seconds often speed up to within one order of magnitude, and those which otherwise require approximately 10000 seconds often speed up to approaching two orders of magnitude. There are several examples which timeout without hints, yet which complete with hints. This is a promising result, as the practical need to improve runtimes of complex, if not "otherwise unsolvable," problems is at the forefront of industrial relevance.

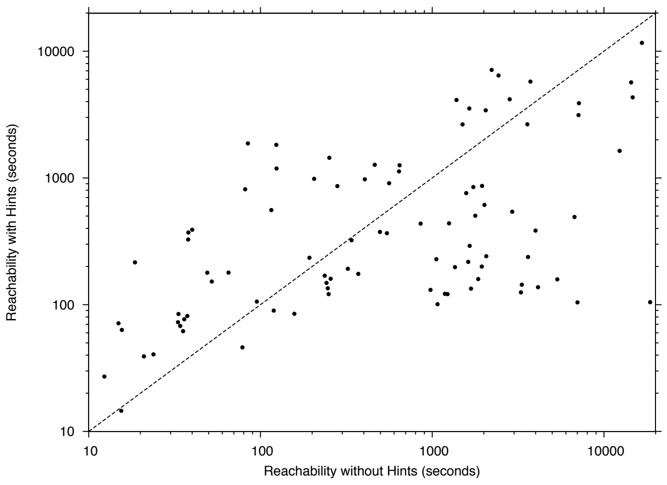


Fig. 1. Reachability Computation Runtime with vs. without Hints

Note that the memory plot exhibits a fair amount of clustering of data points, caused by thresholds at which dynamic

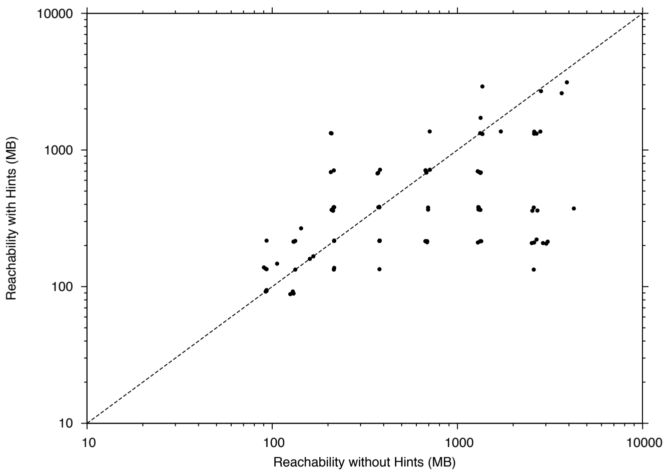


Fig. 2. Reachability Computation Memory with vs. without hints

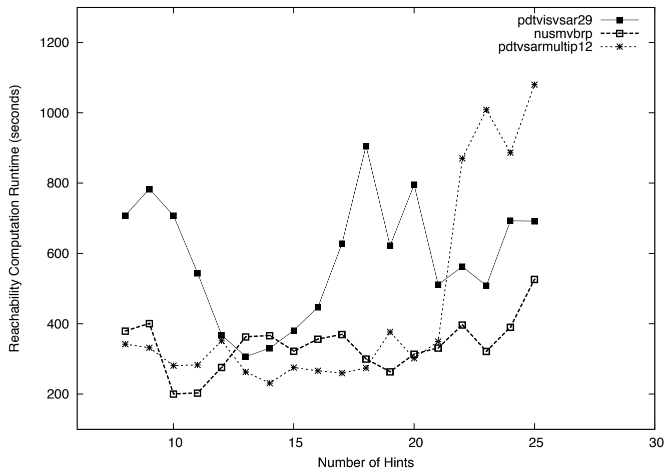


Fig. 3. Impact of number of hints on runtime

variable ordering is invoked. Similarly to the runtime analysis, there are frequent benefits of one to two orders of magnitude for more complex problems, though some penalties primarily for simpler problems.

The observation that hints often entail an overhead for simpler problems prompts the question of whether the introduction of hints should be delayed until a larger threshold. We performed significant experimentation to assess the validity of this strategy and found that delaying the onset of hints almost uniformly hurt complex problems. Invoking hints when the reached set has grown to millions of BDD nodes often requires more expensive dynamic variable ordering calls, and applying a large number of hints at that point degrades performance. Adjusting our hint heuristics to improve performance of simpler problems would compromise performance on complex problems, where hints yield the biggest advantage. We also note that in an industrial-strength multi-engine verification flow, slowdown of simpler problems is not as serious of a concern since within that runtime, one likely would have spent comparable resources trying various alternate algorithms such as bounded model checking and IC3.

Figure 3 illustrates the impact of number of generated hints on runtime. In the experiment, reachability analysis was performed varying the number of hints from 8 to 25. These experiments demonstrate that it is disadvantageous to use too few or too many hints. With too few, the hints do not adequately simplify image computation, while with too many there is too large of an overall increase in the number of computed images. Recall from Algorithm 2 that we use a parameter *var_limit* to limit the maximum number of literals that may be included in a hint. Practically, we have found it useful to bound this parameter based upon netlist size (a percentage of the total number of inputs and state elements) to preclude introducing too many hints for smaller problems.

To justify the importance of highly-tuned reachability engine in a state-of-the-art verification tool, we ran light-weight logic optimization techniques followed by our implementation of IC3 [4], interpolation [3], and k-step unique-state induction engines [24], which are all highly-tuned and competitive with the best academic solvers. Of the 92 benchmarks, 11 resulted in counterexamples for all their properties hence the SAT-based techniques terminated upon finding these counterexamples, whereas we disabled early-termination in our reachability engine for these experiments. We thus omit these 11 from the following experiments. We illustrate the runtimes for the remaining 81 benchmarks using reachability without hints, reachability with hints, and the three SAT-based techniques mentioned above, in Table I. The runtime for the technique which solves most quickly is shown in bold.

Note that 5 benchmarks are solved most quickly using reachability with hints, whereas 14 are solved most quickly using reachability without hints. This collectively represents 23.4% of the benchmarks which are solved more quickly using BDD-based reachability than SAT-based techniques, often by orders of magnitude. The converse is not surprisingly true as well; the SAT-based techniques inherently reason about the design in an abstract manner vs. precisely computing the reachable states, often resulting in much faster runtimes. If we preceded reachability computation by abstraction techniques such as localization [8] or phase abstraction [10], or sequential reductions such as redundancy removal [9] or retiming [11], this would have enabled reachability computation on a larger fragment of the benchmark suite, and have narrowed the precise vs. abstract penalty imposed by these experiments. IC3 solves 27 most quickly (33.3%), and induction solves 35 (43.2%)most quickly, where we broke ties in favor of induction given the maturity and simplicity of that technique. Interpolation was somewhat surprisingly not the winning engine in any of these benchmarks. While we have found IC3 to very often outperform interpolation in practice, there are industrial cases where interpolation is the winning technique.

To further emphasize the role of reachability analysis and hints, we note the following.

1. We only included examples for which hints were generated in these experiments, thus omitted numerous easy wins for reachability in this benchmark suite.

2. Reachability with hints solved all these benchmarks, whereas reachability without hints has 3 timeouts, IC3 has 13, and interpolation and induction each have 41.

3. Reachability using hints outperformed reachability without hints in 46 of these examples (56.8%). As per Figure 1, hints offers greater benefits for more complex benchmarks; if we increase the timeout period, our practical experience is that hints and BDD-based techniques overall play a larger role.

4. In a state-of-the verification tool, lighter-weight algorithms are often leveraged with a moderate resource limit before heavier-weight techniques. If we discount benchmarks solvable within 10 seconds, only 29 of these benchmarks remain: 19 are solved most quickly using reachability (65.5%), 7 using IC3 (24.1%), and 3 using induction (10.3%).

5. Cumulative runtime for reachability with hints is much lesser than for the other techniques while counting timeouts at 4 hours, and even outperforms reachability without hints by a factor of 1.77 when discounting the 3 timeouts for the latter.

Benchmark Name	Inputs / Ands / Registers	Reachability w/o Hints	Reachability with Hints	IC3	Interpolation	Induction
6s4	209 / 2448 / 201	**405.4** (3918)	976.1 (7738)	TO	TO	6081.3
6s48	72 / 796 / 66	7151.8 (17)	**3884.1** (31)	TO	TO	TO
6s48p0	72 / 795 / 66	2434.8 (17)	10620.2 (57)	TO	TO	**430.4**
6s52	35 / 1226 / 207	**65.1** (52)	179.1 (245)	TO	TO	TO
6s53	35 / 1228 / 207	**115.7** (259)	557.6 (1052)	TO	TO	TO
bjrb07amba10andenv	23 / 62516 / 58	**36.0** (41)	76.8 (69)	240.1	TO	TO
bjrb07amba7andenv	17 / 22312 / 45	**15.6** (33)	63.2 (98)	26.2	TO	TO
bjrb07amba9andenv	21 / 45216 / 52	**33.3** (41)	84.5 (188)	75.1	TO	TO
boblivea	5 / 540 / 102	7117.3 (49)	3130.9 (104)	**8.0**	TO	TO
boblivear	5 / 321 / 77	2220.7 (49)	7109.3 (119)	**68.8**	7638.8	TO
eijkbs1512	29 / 817 / 123	TO (544)	5675.1 (1300)	**1.3**	TO	TO
eijkbs3330	37 / 1407 / 166	TO (4)	11637.8 (48)	**23.2**	TO	TO
intel055	222 / 3847 / 124	561.5 (24)	908.6 (75)	**16.6**	TO	TO
intel059	280 / 1955 / 140	646.2 (24)	1257.5 (83)	**13.7**	TO	TO
intel063	288 / 1773 / 240	34.1 (7)	67.9 (19)	**0.6**	0.7	TO
nusmvbrp	11 / 378 / 51	1952.2 (57)	864.8 (158)	**2.2**	3492.6	TO
nusmvdme1d3multi	54 / 236 / 61	TO (38)	**104.6** (270)	TO	TO	TO
nusmvqueue	82 / 1200 / 84	**462.4** (45)	1270.0 (136)	5246.5	TO	TO
pdtfifo1to0	6 / 860 / 142	**251.6** (62)	1440.6 (398)	5517.9	TO	TO
pdtpmsbufferalloc	6 / 477 / 66	78.6 (31)	**46.0** (57)	TO	TO	TO
pdtpmseisenberg	3 / 1765 / 125	544.9 (90)	**366.2** (223)	TO	TO	TO
pdtpmsfpmult	17 / 929 / 166	49.0 (7)	178.6 (37)	**1.0**	14176.8	TO
pdtpmsgigamax	22 / 681 / 85	6.9 (8)	22.5 (37)	**0.3**	4.5	TO
pdtpmsns2	16 / 1742 / 278	339.1 (16)	322.7 (38)	**56.2**	TO	TO
pdtpmstimeout	10 / 922 / 80	**12.3** (28)	27.1 (64)	TO	TO	TO
pdtswvibs8x8p1	9 / 1039 / 96	81.5 (83)	813.3 (523)	5.1	47.3	**4.6**
pdtswvqis10x6p1	7 / 1609 / 92	124.0 (99)	1187.1 (487)	**81.0**	TO	TO
pdtswvqis10x6p2	7 / 1771 / 88	**84.5** (99)	1871.4 (489)	TO	TO	TO
pdtswvqis8x8p1	9 / 1685 / 98	**18.6** (79)	215.5 (325)	48.6	2492.3	6612.2
pdtswvqis8x8p2	9 / 1866 / 94	**37.9** (79)	326.4 (349)	TO	TO	TO
pdtswvrod6x8p1	9 / 1314 / 74	40.0 (132)	**39.2** (748)	100.4	TO	TO
pdtswvrod6x8p2	9 / 1331 / 70	**38.0** (132)	371.0 (772)	TO	TO	TO
pdtswvroz10x6p1	7 / 926 / 73	52.1 (87)	152.1 (367)	**3.5**	3413.2	27.6
pdtswvroz10x6p2	7 / 941 / 73	192.9 (87)	234.6 (391)	**13.3**	TO	2262.8
pdtswvsam6x8p4	9 / 2003 / 116	1385.7 (69)	4125.2 (453)	TO	TO	**264.9**
pdtswvtma6x4p2	5 / 457 / 42	37.5 (60)	81.4 (159)	92.4	TO	**8.3**
pdtswvtma6x4p3	5 / 459 / 42	**14.9** (60)	24.1 (164)	918.4	TO	42.2
pdtswvtma6x6p1	7 / 640 / 58	205.4 (60)	984.1 (235)	48.6	904.2	**6.7**
pdtswvtma6x6p2	7 / 607 / 58	280.3 (60)	861.9 (242)	1002.1	TO	**49.0**
pdtvisns3p00	21 / 1210 / 100	371.3 (25)	174.8 (60)	**3.6**	TO	TO
pdtvisns3p01	21 / 1220 / 100	157.7 (25)	84.7 (83)	**5.6**	TO	TO
pdtvisns3p02	21 / 1206 / 100	322.8 (25)	191.8 (130)	**3.0**	TO	TO
pdtvisns3p03	21 / 1200 / 100	249.4 (25)	121.1 (43)	**2.2**	TO	TO
pdtvisns3p04	21 / 1183 / 100	246.5 (25)	134.6 (146)	**3.9**	TO	TO
pdtvisns3p05	21 / 1179 / 100	95.3 (25)	105.7 (150)	**3.3**	TO	TO
pdtvisns3p06	21 / 1181 / 100	256.4 (25)	159.9 (42)	**6.7**	TO	TO
pdtvisns3p07	21 / 1190 / 100	236.6 (25)	169.2 (78)	**3.9**	TO	TO
pdtvisns3p08	21 / 1176 / 100	242.4 (25)	148.5 (127)	**0.8**	TO	TO
pdtvisns3p09	21 / 1178 / 100	119.5 (25)	89.7 (90)	**0.9**	TO	TO
pdtvissoap1	11 / 1510 / 124	23.8 (46)	40.5 (77)	**1.7**	TO	TO
pdtvissoap2	11 / 1548 / 124	21.0 (46)	39.0 (118)	**1.2**	149.7	TO
pdtvisvsar27	17 / 898 / 62	1622.1 (36)	217.5 (192)	0.1	0.3	**0.1**
pdtvisvsar29	17 / 1081 / 61	3994.4 (36)	383.9 (111)	120.2	5049.6	**0.3**
pdtvsarmultip	17 / 1473 / 77	2922.6 (36)	541.6 (116)	65.2	1891.5	**0.8**
pdtvsarmultip00	17 / 860 / 61	1683.8 (36)	133.8 (139)	0.1	0.2	**0.1**
pdtvsarmultip03	17 / 873 / 61	1942.7 (36)	199.7 (118)	0.1	0.1	**0.1**
pdtvsarmultip04	17 / 873 / 61	3285.1 (36)	125.1 (237)	0.1	0.1	**0.1**
pdtvsarmultip05	17 / 850 / 61	855.9 (36)	914.6 (187)	0.5	0.3	**0.1**
pdtvsarmultip06	17 / 862 / 61	7017.1 (36)	104.3 (261)	0.2	0.2	**0.1**
pdtvsarmultip07	17 / 890 / 61	4134.4 (36)	137.3 (94)	0.4	0.4	**0.1**
pdtvsarmultip08	17 / 857 / 61	3584.0 (36)	2650.1 (478)	0.1	0.2	**0.1**
pdtvsarmultip09	17 / 852 / 61	1653.3 (36)	291.2 (180)	0.1	0.2	**0.1**
pdtvsarmultip10	17 / 852 / 61	2065.2 (36)	241.1 (131)	0.4	0.3	**0.1**
pdtvsarmultip11	17 / 870 / 62	1252.5 (36)	438.4 (132)	0.1	0.1	**0.1**
pdtvsarmultip12	17 / 866 / 61	1229.7 (36)	121.1 (214)	0.1	0.1	**0.1**
pdtvsarmultip13	17 / 869 / 64	3613.9 (36)	237.8 (173)	0.1	0.1	**0.1**
pdtvsarmultip14	17 / 900 / 61	1074.4 (36)	100.9 (170)	0.1	0.1	**0.1**
pdtvsarmultip15	17 / 880 / 61	1057.6 (36)	228.4 (124)	0.1	0.1	**0.1**
pdtvsarmultip17	17 / 879 / 63	3326.2 (36)	143.7 (121)	0.1	0.1	**0.1**
pdtvsarmultip19	17 / 876 / 62	977.3 (36)	130.7 (109)	0.1	0.1	**0.1**

continued on next page

Benchmark Name	**Inputs / Ands / Registers**	**Reachability w/o Hints**	**Reachability with Hints**	**IC3**	**Interpolation**	**Induction**
pdtvsarmultip21	17 / 874 / 62	496.3 (36)	375.0 (254)	0.1	0.1	**0.1**
pdtvsarmultip22	17 / 846 / 62	1356.7 (36)	197.6 (115)	0.1	0.1	**0.1**
pdtvsarmultip23	17 / 865 / 62	1852.7 (36)	159.1 (121)	0.1	0.1	**0.1**
pdtvsarmultip24	17 / 861 / 62	5350.6 (36)	158.4 (170)	0.1	0.1	**0.1**
pdtvsarmultip26	17 / 865 / 62	2016.4 (36)	612.8 (234)	0.1	0.1	**0.1**
pdtvsarmultip27	17 / 882 / 62	1186.4 (36)	121.8 (220)	0.1	0.3	**0.1**
pdtvsarmultip29	17 / 1064 / 61	1735.2 (36)	1747.5 (176)	802.6	2636.6	**0.5**
pdtvsarmultip31	17 / 1002 / 62	1781.9 (36)	502.7 (167)	0.1	0.1	**0.1**
pdtvsarmultip32	17 / 983 / 61	6739.3 (36)	491.7 (179)	25.5	86.0	**0.2**
pj2009	304 / 7498 / 269	3734.3 (31)	5748.3 (159)	**4.3**	20.4	TO
sm98a7multi	82 / 3337 / 89	12346.1 (37)	1632.9 (161)	2.8	1.4	**1.1**
Cumulative		158558.6	**82707.6**	201872.0	632409.5	606195.3

TABLE I. Runtimes for various proof engines. Column 2 provides size of the benchmark after light-weight reductions. Subsequent columns list runtimes in seconds; TO refers to 4-hour timeout. The number in parenthesis in Columns 3 and 4 indicates the number of image computations until fixedpoint or TO.

While points **2** and **3** above are skewed by the selection of benchmarks for which reachability in some form converges, these experiments do emphasize that reachability often outperforms SAT-based techniques, and hints increase the overall robustness of reachability computation.

V. Conclusion

Despite many advances in SAT-based proof techniques, BDD-based reachability remains a critical technology which is able to significantly outperform alternative proof techniques on numerous classes of problems. In this paper, we introduce a novel technique to increase the scalability of reachability computation: automated dynamic netlist-based hint generation. Experiments demonstrate that this approach is able to reduce resources well over an order of magnitude on many complex verification problems, outperforming SAT-based techniques in many cases. These techniques have played a vital role in revitalizing reachability analysis as a core industrial-strength proof technique in our multi-algorithm verification toolsuite.

References

[1] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *VLSI*, pp. 49–58, Aug. 1991.

[2] I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *FMCAD*, Nov. 2000.

[3] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, 2003.

[4] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Jan. 2011.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.

[6] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *ICCAD*, Nov. 2000.

[7] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To split or to conjoin: the question in image computation," in *DAC*, June 2000.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, July 2000.

[9] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *DATE*, 2009.

[10] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[11] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.

[12] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[13] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/alanmi/abc.

[14] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[15] P. A. Beerel, J. R. Burch, and K. L. McMillan, "Sibling-substitution-based BDD minimization using don't cares," *TCAD*, vol. 19, Jan. 2000.

[16] K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *CHARME*, Oct. 1999.

[17] D. Ward and F. Somenzi, "Automatic generation of hints for symbolic traversal," in *CHARME*, Sept. 2005.

[18] D. Ward and F. Somenzi, "Decomposing image computation for symbolic reachability analysis using control flow information," in *ICCAD*, Nov. 2006.

[19] K. Ravi and F. Somenzi, "High-density reachability analysis," in *ICCAD*, Nov. 1995.

[20] Hardware Model Checking Competition 2011. http://fmv.jku.at/hwmcc11.

[21] G. Janssen, "Design of a pointerless BDD package.," in *IWLS*, 2001.

[22] H. Fujii, G. Ootomo, and C. Hori, "Interleaving based variable ordering methods for ordered binary decision diagrams," in *ICCAD*, Nov. 1993.

[23] H. Jin, A. Kuehlmann, and F. Somenzi, "Fine-grain conjunction scheduling for symbolic reachability analysis," in *Tools and Algos. Construction and Analysis of Systems*, April 2002.

[24] N. Eén and N. Sörennson, "Temporal induction by incremental SAT solving," in *Workshop on Bounded Model Checking*, 2003.

Oscillator Verification with Probability One

Chao Yan
Intel

Mark Greenstreet
University of British Columbia

Abstract—**This paper presents the formal verification of start-up for a differential ring-oscillator circuit used in industrial designs. Dynamical systems theory shows that any oscillator must have a non-empty failure; however, it is possible to show that these failures only occur with zero probability. To do so, this paper generalizes the "cone argument" initially presented in [1] and proves the soundness of this generalization. This paper also shows how concepts from analog design such as differential operation can be soundly incorporated into the verification to produce simpler models and reduce the complexity of the verification task.**

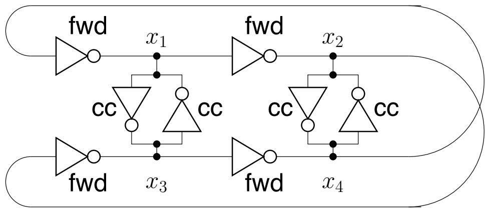


Fig. 1. Ring-Oscillator Example from Rambus

I. Introduction

System-on-Chip (SoC) and analog-mixed-signal (AMS) designs have created new challenges for analog circuit designers. Typical analog design relies heavily upon simulation tools such as HSPICE and Spectre. Long simulation times along with the continuous nature of device parameters, operating conditions and input waveforms mean that simulation tools can only provide partial verification of analog designs. In practice, designers typically focus their simulation efforts on parametric and small-signal sensitivity analysis when the circuit is in or near its intended operating mode. Such analysis can be used to determine the gain and bandwidth of an amplifier, the jitter transfer function of a phase-locked loop, along with finding transistor sizes to optimize a given circuit topology for an objective function formulated in terms of steady-state properties of the circuit. However, simulations cannot show that the circuit will eventually reach its intended operating condition from all possible starting conditions.

This paper presents a rigorous, formal verification that a commonly used differential ring-oscillator circuit correctly starts oscillation with probability 1. As shown in Figure 1, the oscillator consists of two stages, where each stage has a pair of "forward" inverters (labeled fwd in the figure) and a pair of "cross-coupling" inverters (labeled cc). If the forward inverters are much larger than the cross-coupling inverters, then the circuit acts like a ring of four inverters settles to one of two states:

State 1: X1 and X3, are low; and X2 and X4 are high.
State 2: X1 and X3, are high; and X2 and X4 are low. (1)

Conversely, if the cross-coupling inverters are much larger than the forward ones, then the circuit acts like two separate static latches and has four stable states. If the forward and cross-coupling inverters have comparable strength, then the circuit should oscillate in a stable fashion.

In 2008, researcher from Rambus posed the problem of showing that the oscillator circuit shown in Figure 1 starts from all initial conditions for a particular choice of transistor sizes [2]. They described this as a "real-world" problem noting that oscillators of this type had been observed to fail in the test-lab. They posed a further problem of determining the range of transistor sizes for which proper start-up is guaranteed. This paper presents solutions to these problems.

A. Prior Work

Oscillator circuits have been a popular example for applying formal methods to analog circuit verification [3]–[6]. These early papers focused on simple oscillators, such as a tunnel-diode based design, that are not representative of the oscillator circuits used in real VLSI designs. More recently, several groups have reported results for the Rambus oscillator problem described above.

The earliest attempted verification of the oscillator that we have seen [7] predates the formulation by [2] and considers the behaviours of a 128 stage oscillator for a pulse-width modulated voltage regulator. Their "proof" of correct operation assumes differential and periodic operation, and does not consider weak coupling between stages (e.g. due to power supply noise), that could stabilize undesired, higher harmonic modes of oscillation.

A more rigorous approach was taken in [8] which used monotonicity properties of the i_{ds} functions of MOSFETs to reduce the search for DC-equilibria in a Rambus ring oscillator with an arbitrary, even number of stages to a one-dimensional search, regardless of the number of stages in the oscillator. They then used standard, small-signal analysis techniques to determine if any of these equilibria are stable. If an oscillator circuit had no stable DC equilibria, it was deemed free from DC lock-up. The authors noted that their proof did not rule out other behaviours such as higher harmonic oscillations or chaotic behaviours.

Several subsequent papers have also treated the verification problem as one of ruling out the existence of DC equilibria. For example, Tiwari *et al*. [9], [10] used a SAT solver to identify DC equilibria. To find *stable* equilibria, they added constraints that at least one node of the circuit must be within 0.2 volts of power or ground. They did not state how they had arrived at these extra constraints or whether or not they can be shown to be sound. Steinhorst *et al.* [11] presented a particle filtering approach and compared it with a model-checking method. The correctness condition for the model-checking was lack of stable DC-equilibria. While higher-harmonic oscillations or chaotic behaviours were not considered, they presumably would show up in the visualizations if suitable particles were included in the state-space sampling. Zaki *et al.* [12] presented an approach where the "pencil-and-paper" analysis from [8] was automated using HySAT [13] and Matlab toolboxes for interval arithmetic and matrix pseudo-spectrum calculations.

Little *et al.* [14] showed that trajectories in a neighbourhood of the nominal periodic trajectory for the oscillator remain close to that nominal trajectory. This replaces the small-signal analysis of traditional analog design with linear hybrid Petri net (LHPN) model checking and confirms the stability of the desired oscillating behaviour. As the analysis only considers a portion on the state space near the desired trajectory, it does not verify proper start-up for all initial conditions.

B. Contributions

This paper combines analytical techniques based on dynamical systems theory with reachability tools to present the first verification of the Rambus oscillator problem that actually addresses the question posed by Jones *et al:* "Will the oscillator start up from all initial conditions?" In Section II we consider the dynamics of *any* oscillator that is modeled by non-linear differential equations and show that it must have some set of initial conditions for which the circuit fails to oscillate. However, this failure set can be *negligible*, i.e. have zero probability. We present a generalization of the cone argument from [1] to verify that the failure set has zero probability, and thus that the oscillator starts with a probability of one. We also introduce a symmetry reduction method that allows us to exploit the differential operation of the oscillator in a formal verification context. Section IV describes our implementation of the verification method using Matlab and Coho [15]. Section V presents the results of verifying the oscillator circuit with these methods.

II. No Perfect Oscillator

This section shows that no physically plausible oscillator starts from all initial conditions.

A. Dynamical Systems and Oscillators

We assume that a circuit, such as an oscillator, is modeled by a system of ordinary differential equations. If the model has d variables, states of the circuit correspond to points in $\mathbb{R}^d$. The model includes a function, $f : \mathbb{R}^d \to \mathbb{R}^d$ that is the *time derivative* of the system: for state $\mathbf{x} \in \mathbb{R}^d$, $\dot{\mathbf{x}} = f(\mathbf{x})$ is the time-derivative of the system in state $\mathbf{x}$. For $\mathbf{x}_0 \in \mathbb{R}^d$, the *initial value problem* is to find a function $\mathbf{x} : \mathbb{R}^+ \to \mathbb{R}^d$ such that for all $t \geq 0$, $\frac{d}{dt}\mathbf{x}(t) = f(\mathbf{x}(t))$ and $\mathbf{x}(0) = \mathbf{x}_0$. Let $Q \subseteq \mathbb{R}^d$ be a closed set. Q is invariant with respect to f if all trajectories that start in Q remain in Q forever. To show that Q is invariant with respect to f, it is sufficient to show that for every $\mathbf{x} \in Q$ there is an $\varepsilon > 0$ such that $\mathbf{x} + \varepsilon f(\mathbf{x}) \in Q$. We impose two restrictions on f:

R1: There is a $Q \subseteq \mathbb{R}^d$ and some $K \in \mathbb{R}$ such that Q is invariant with respect to f and for every $\mathbf{x} \in Q$, $\|f(\mathbf{x})\| < K$.

R2: f is $\mathscr{C}^1$ in Q. This means that $f(\mathbf{x})$ is differentiable with respect to the components of $\mathbf{x}$, and these derivatives are continuous.

These two conditions guarantee the existence and uniqueness of solutions to the initial value problem for f and any $\mathbf{x}_0 \in Q$ (see [16, chap. 8.3]). We can define a function $\Phi_f(\mathbf{x}_0, t)$ such that if $\mathbf{x}$ is the solution to the initial value problem for f with $\mathbf{x}(0) = \mathbf{x}_0$, then $\mathbf{x}(t) = \Phi_f(\mathbf{x}_0, t)$. Given restrictions R1 and R2, $\Phi_f(\mathbf{x}_0, t)$ is a $\mathscr{C}^1$ function with respect to $\mathbf{x}_0$ and t for any $\mathbf{x}_0 \in Q$ and $t \geq 0$ (see [16, chap. 8.4]). We extend Φ_f to sets in the natural way: if $X \subseteq \mathbb{R}^d$, then $\Phi_f(X, t) = \{\mathbf{x}_2 | \exists \mathbf{x}_1 \in X.\, \mathbf{x}_2 = \Phi_f(\mathbf{x}_1, t) |\}$.

We assume that any physically plausible oscillator can be modeled by an ODE with f and Q satisfying restrictions R1 and R2. The requirement that f is $\mathscr{C}^1$ follows from the smoothness of the underlying physical models for electric fields, charge distributions, etc. The requirement of the existence of the set Q is satisfied because VLSI circuits generally have node voltages that are bounded by the voltages of ground and the power supply or that have limited excursions beyond these power supply voltages.

We now define "oscillation." If there is a $\mathbf{x}_0 \in Q$ and a $P > 0$ such that $\Phi_f(\mathbf{x}_0, P) = \mathbf{x}_0$, and for all $0 < t < P$, $\mathbf{x}(t) \neq \mathbf{x}_0$, then f has a solution with period P. In this case, we write $\Gamma_{f,\mathbf{x}_0} = \{\mathbf{x} | \exists t \in [0, P].\, \mathbf{x} = \Phi_f(\mathbf{x}_0, t)\}$ to denote the set of points in this periodic orbit. It is straightforward to show $\forall t > 0.\, \Phi_f(\Gamma, t) = \Gamma$. Let $J = Jac\, \Phi_f(\mathbf{x}_0, P)$, i.e., J is the matrix of partial derivatives of $\Phi_f(\mathbf{x}_0, P)$ with respect to $\mathbf{x}_0$. If J has $d-1$ eigenvalues with magnitude less than 1, then the periodic solution for $\mathbf{x}_0$ is a *periodic attractor* [16, Theorem 13.2]. We say that a system is an oscillator with period P if it has a periodic attractor with period P.

B. Oscillator Start-Up

First consider the set of possible initial states. Labeling one terminal of the power supply as "ground" and the other as "V_{dd}" is simply a designer convention. Depending on circuit details, the node voltages on power-up may be arbitrary values. Rather than trying to analyse the circuit in detail, we simply assume that each node has an arbitrary initial voltage in $[V_{lo}, V_{hi}]$; typically V_{lo} is ground or close to ground, and V_{hi} is close to V_{dd}. Let $X_0 = [V_{lo}, V_{hi}]^d$ denote the set of initial node voltages. Because X_0 contains all reachable states of the circuit, we assume $\gamma \subseteq X_0 \subset Q$, where γ is the desired periodic attractor of the oscillator.

We can now describe an ideal oscillator.

A d-dimensional dynamical system with time-derivative function f is an *ideal oscillator* iff

- **The system is physically plausible:** There is a set $Q \subseteq \mathbb{R}^d$ such that f and Q satisfy conditions R1 and R2.
- **Periodic behavior:** The system has a periodic attractor. Let Γ be the orbit associated with this attractor.
- **Start up:** There is a convex set $X_0 \subseteq \mathbb{R}^d$ of initial states such that $\Gamma \subseteq X_0 \subseteq Q$ and for every point $\mathbf{x}_0 \in X_0$ and every $\varepsilon > 0$, there is a $t > 0$ and a point $\mathbf{x}_1 \in \Gamma$ such that $\|\mathbf{x}_1 - \Phi(\mathbf{x}_0, t)\| < \varepsilon$.

The first two conditions were described in the previous section. The last condition states that the set of initial states must contain the periodic orbit as described above, and that for any initial state, the trajectories emanating from that state must eventually be arbitrarily close to the periodic orbit. The requirement that this initial set be convex reflects the topological properties of sets such as $[V_{lo}, V_{hi}]^d$ described above. We believe that this definition of an ideal oscillator captures the notion of the oscillator starting from all initial conditions requested in [2].

Theorem 1. *There is no ideal oscillator.*

Proof: This follows directly from the property that solutions of ODEs that satisfy properties R1 and R2 are continuous in their initial conditions. Thus, the topology of the initial set, X_0, is preserved by $\Phi_f(X_0, t)$. However, any small neighborhood of a periodic attractor must have genus 1 (be "torus-like") whereas the set of initial states has genus 0 (i.e. it is "sphere-like"). Thus, it is not that case that all initial conditions lead to trajectories that are arbitrarily close to the desired attractor. This establishes the claim. ■

III. Verification Outline

Our verification proceeds in three main phases:

- **Differential Operation** The oscillator shown in Figure 1 is a differential design: nodes X1 and X3 form a "differential pair" and likewise for nodes X2 and X4. The first phase of the verification shows that each of these differential pairs can be treated as a single signal.
- **Escape from the Failure Set** As shown in Section II, for any oscillator, there must be initial conditions from which it does not properly start. The second phase of the verification shows that this occurs with probability zero.
- **Proper Oscillation** The first two phases show that most initial conditions lead to a fairly small subset of the full state space. In the final phase, we use existing reachability methods to show that the oscillator starts up properly from the region.

This section describes the dynamical systems issues associated with each of these phases. Section IV describes our verification method based on these observations.

We model the oscillator circuit from Figure 1 using non-linear ordinary differential equations (ODEs) of the form:

$$\dot{\mathbf{x}} \quad = \quad f(\mathbf{x}) \tag{2}$$

where $\mathbf{x}$ is a vector of node voltages, $\dot{\mathbf{x}}$ is the vector of time derivatives for these voltages, and f is the function modeling the non-linear dynamics of this circuit. Let d be the dimensionality of $\mathbf{x}$. We assume that f is C^1 which guarantees that Equation 2 has a unique solution for any initial state, $\mathbf{x}(0)$. For simplicity, we model the system as being autonomous (no inputs or outputs). Inputs (e.g. to model VCO control inputs, power supply noise), can be modeled by giving f additional parameters, i.e. $f(\mathbf{x}, \mathbf{in})$.

A. Differential Behaviour

Nodes X1 and X3 in the oscillator from Figure 1 form a "differential pair" and likewise for nodes X2 and X4. Let x_i denote the voltage on node Xi. The *differential component* of the differential pair is $x_1 - x_3$, and $x_1 + x_3$ is the *common mode* component. When the oscillator is operating properly, the common mode components are roughly constant and the oscillation is manifested in the differential components. Let V_0^+ be the nominal value for the common mode components. We show that for a relatively small V_{err} if $|x_1 + x_3 - V_0^+| > V_{err}$, then $\frac{d}{dt}(x_1 + x_3)$ and $(x_1 + x_3 - V_0^+)$ have opposite signs. This shows that that the common mode component for nodes X1 and X3 converges to within V_{err} of the nominal value. Likewise for nodes X2 and X4.

B. Escape from the Failure Set

Theorem 1 shows that there is no perfect oscillator. For the Rambus ring-oscillator, there is an equilibrium point, $\mathbf{x}_{fail}$, i.e. a point where $\dot{\mathbf{x}} = 0$, and there is a manifold, X_{fail} such that

$$\forall \mathbf{x} \in X_{fail}. \lim_{t \to \infty} \|\Phi_f(\mathbf{x}, t) - \mathbf{x}_{fail}\| = 0 .$$

Thus, direct application of continuous state-space model checkers (e.g. [3], [17]) to the oscillator start-up problem will identify regions where trajectories might stay forever. Because we cannot show that the set of failure states is empty, we must settle for showing that it is *negligible* (i.e. occurs with probability zero). This is sufficient in practice, as designers are not worried about a design that fails with probability zero.

For intuition, consider an oscillator where all inverters are identical. We define V_{eq} as the voltage that can be applied to the input of the inverter such that the output settles to the same voltage. When all of the inverters are identical, $\mathbf{x}_{fail}$ is the point at which all node voltages are V_{eq}. Furthermore, any trajectory starting at a point where $x_1 = x_3$ and $x_2 = x_4$ converges to $\mathbf{x}_{fail}$; thus, such points are in X_{fail}.

Using existing reachability methods, we can find a small region, U_{fail}, that contains the point $\mathbf{x}_{fail}$. Furthermore, we can show that if an oscillator starts any point where each node has a voltage in the interval $[0, V_{dd}]$, then within bounded time, the oscillator state will either be in U_{fail}, or it will be in a region where we can show convergence to the desired periodic orbit.

We will show that the set of failing trajectories is sufficiently small as to ensure that the oscillator fails to start with a probability of zero. As in the previous section, we write $\mathbb{R}^d$ to denote the phase space. We will avoid a detailed treatment of measure theory (see [18]) by noting that when we say that $B \subseteq \mathbb{R}^d$ is measurable, we mean that it has a well-defined d-dimensional "volume" (i.e. it is Lesbesgue measurable), and we write $|B|$ to denote this volume (i.e. measure). We write $\mu(B)$ to denote the probability that the initial state of the oscillator is in B. Our assumption that μ is smooth (i.e. absolutely continuous) means that if $|B|$ is zero, then $\mu(B)$ is zero as well. For example, let

$$B \quad = \quad \{(x_1,x_2,x_3,x_4) \mid (x_1 = x_3) \wedge (x_2 = x_4)\}$$

i.e. the plane described above. Because this plane has zero volume, $|B| = 0$, and by our smoothness assumption, $\mu(B) = 0$ as well.

Let U be a bounded, measurable subset of $\mathbb{R}^d$. We define

$$\begin{array}{rcl} \mathsf{escape}_f(\mathbf{x},U) & = & \exists t \in \mathbb{R}^+.\ \Phi_f(\mathbf{x},t) \notin U \\ \mathsf{trapped}_f(U) & = & \{\mathbf{x} \in U \mid \neg\mathsf{escape}_f(\mathbf{x},U)\} \end{array}$$

For any $U \subseteq \mathbb{R}^+$, and any $t \in \mathbb{R}$, $|U| = 0 \Leftrightarrow |\Phi_f(U,t)| = 0$. Thus, it suffices to show that $|\mathsf{trapped}_f(U_{fail})| = 0$. The next theorem presents conditions that ensure $\mu(\mathsf{trapped}_f(U)) = 0$.

Theorem 2. *Let μ be a smooth probability measure over $\mathbb{R}^d$. Let U be a bounded, measurable subset of $\mathbb{R}^d$, and $f : \mathbb{R}^d \to \mathbb{R}^d$ be bounded and C^1 in U. If there is a matrix $H \in \mathbb{R}^{d\times d}$ such that at least one eigenvalue of H has a positive real part, and $k > 0$ such that for all $\mathbf{x}_1, \mathbf{x}_2 \in U$:*

$$\begin{array}{rl} & (\mathbf{x}_2 - \mathbf{x}_1)^T H (\mathbf{x}_2 - \mathbf{x}_1) \ > \ 0 \\ \Rightarrow & (\mathbf{x}_2 - \mathbf{x}_1)^T H (f(\mathbf{x}_2) - f(\mathbf{x}_1)) \ > \ k(\mathbf{x}_2 - \mathbf{x}_1)^T H (\mathbf{x}_2 - \mathbf{x}_1)\,, \end{array}$$

then $\mu(\mathsf{trapped}_f(U)) = 0$.

Proof: Assume that $\mathsf{trapped}_f(U) \neq \emptyset$ as the other case is trivial. Let $\rho_{\max}$ be the maximum real part of any eigenvector of H. Let $\mathbf{u}$ be a unit vector such that $u^T H u = \rho_{\max}$. Let $\mathbf{x}_0$ be any point in $\mathsf{trapped}_f(U)$, and $\alpha \in \mathbb{R}$ such that $\alpha > 0$ and $\mathbf{x}_0 + \alpha\mathbf{u} \in U$. We'll define $\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{u}$.

We now show $\mathbf{x}_1 \notin \mathsf{trapped}_f(U)$. Consider two trajectories,

$$\begin{array}{rcll} \eta_0(t) & = & \Phi_f(\mathbf{x}_0,t), & \text{the trajectory that starts at } \mathbf{x}_0 \\ \eta_1(t) & = & \phi_f(\mathbf{x}_1,t), & \text{the trajectory that starts at } \mathbf{x}_1 \end{array}$$

Note that both trajectories start in U. We'll show that these two trajectories diverge, and thus at most one of them can remain in U. Let

$$w(t) \quad = \quad (\eta_1(t) - \eta_0(t))^T H (\eta_1(t) - \eta_0(t))$$

We claim that for $t \geq 0$, $w(t) \geq \alpha^2 \rho_{\max} e^{kt} > 0$. First note that $w(0) = \alpha^2 \rho_{\max}$ which satisfies the claim (at $t = 0$). Both $w(t)$ and $\alpha^2 \rho_{\max} e^{kt}$ are continuous functions of t. Thus, if the claim were ever to be violated, there would have to be a value of t for which $w(t) = \alpha^2 \rho_{\max} e^{kt}$ and $\frac{d}{dt} w(t) < \frac{d}{dt} \alpha^2 \rho_{\max} e^{kt}$. For the sake of contradiction, let t be such a time. Then

$$\begin{array}{rcl} \frac{d}{dt} w(t) & = & (\eta_1(t) - \eta_0(t))^T H (f(\eta_1(t)) - f(\eta_0(t))) \\ & > & k(\eta_1(t) - \eta_0(t))^T H (\eta_1(t) - \eta_0(t)) \\ & = & kw \quad = \quad k\alpha^2 \rho_{\max} e^{kt} \quad = \quad \frac{d}{dt} \alpha^2 \rho_{\max} e^{kt} \end{array}$$

But this shows that $\frac{d}{dt} w(t) > \frac{d}{dt} \alpha^2 \rho_{\max} e^{kt}$, a contradiction. Thus, $w(t) \geq \alpha^2 \rho_{\max} e^{kt}$ as claimed.

Because, $w(t) \geq \alpha^2 \rho_{\max} e^{kt}$, $\|\eta_1(t) - \eta_0(t)\|$ must diverge as $t \to \infty$. By assumption, $\eta_0(t)$ stays in U, and U is bounded. Therefore, $\eta_1(t)$ must exit U.

We have shown that for any point $\mathbf{x}_0 \in \mathsf{trapped}_f(U)$, all points in the cone defined by H whose apex is at $\mathbf{x}_0$ must escape from U. This shows that $\mathsf{trapped}_f(U)$ must have lower dimension than the full space. Thus, $|\mathsf{trapped}_f(U)| = 0$, and therefore $\mu(\mathsf{trapped}_f(U)) = 0$ as claimed. ■

Note: Theorem 2 was based on the cone argument from [1]. The present theorem generalizes the result from [1] to systems of arbitrary dimensions and whose Jacobian matrices have complex eigenvalues. The conditions for Theorem 2 are slightly stronger than those from [1] (for the systems where the latter applies) – this is mainly for simplicity.

C. Proper Oscillation

For the trajectories under consideration after the first two steps, the common mode components of both differential signal pairs are within V_{err} of V_0^+. This allows the differential equation model from Equation 2 to be rewritten as a *differential inclusion* [19]:

$$\dot{\mathbf{u}} \quad \in \quad F(\mathbf{u}) \tag{3}$$

where $\mathbf{u}$ is the vector $(\sqrt{2}/2)[x_1 - x_3, x_2 - x_4]$. By using an inclusion, F accounts for *all* values of the common mode components in $[V_0^+ - V_{err}, V_0^+ + V_{err}]$. Reducing the four-dimensional state space of the original problem to a two-dimensional space makes the exploration of trajectories from all remaining start conditions straightforward.

By showing that all such trajectories lead to an oscillation in the fundamental mode, we solve the first part of the challenge problem from [2]: we show that for a particular choice of transistor sizes, the circuit will start oscillation from all initial conditions except for a set of zero measure. Section V provides a brief description of how these methods can be extended to establish a range of transistor sizes for which the oscillator will start with probability one.

IV. Implementation

This section describes our implementation of the verification techniques described in the previous section. We construct an ODE model for the ring oscillator circuit using standard, modified nodal analysis. We obtain drain-to-source current data by tabulating HSPICE outputs and fitting piece-wise quadratic functions to this tabulated data. The resulting errors are less than 1%; thus, our transistor models closely match those used by practicing circuit designers in industry.

A. Differential Operation

This verification phase starts by changing the coordinate system to one based on the differential and common mode representation of signals.

Let $\mathbf{u}$ be the circuit state in "differential" coordinates:

$$\begin{array}{rcl} \mathbf{u} & = & M^{-1}\mathbf{x} \\ M & = & \frac{\sqrt{2}}{2} \left[\begin{array}{rrrr} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{array} \right] \end{array} \tag{4}$$

We assume each of nodes X1, X2, X3 and X4 can independently have initial voltages anywhere in $[0,1.8]$V. Thus, the differential components, u_1 and u_2, are initially in $[-0.9\sqrt{2}, +0.9\sqrt{2}]$, and the common mode components, u_3 and u_4, are initially in $[0, 1.8\sqrt{2}]$.

To establish differential operation, we divide the range of each component u_i of $\mathbf{u}$ into m intervals, creating m^4 cubes. We construct a graph, $G = (V,E)$ to represent the reachability relationship between these cubes. Let $v_{i,j,k,\ell}$ be a vertex corresponding to the i^{th} interval for u_1, the j^{th} interval for u_2 and so on. There is an edge from v to w if f allows a flow out of the cube for v directly into the cube for w, and there is a self-loop for v if each component of f is zero somewhere in v. If a vertex of G has no incoming edges, then any trajectory that starts in the corresponding cube will eventually leave that cube, and no trajectories will ever enter the cube. Such a cube can be eliminated from further consideration. Thus, we only need to consider cubes whose vertices are members of cycles. These vertices can be identified in $O(V+E) = O(m^4)$ time. With a direct implementation of this computation, constructing G dominates the entire time for verifying the oscillator.

To obtain a more efficient computation, we first note that the goal is to establish differential operation. It is sufficient to project the vertices of V onto the common-mode components of the differential signals and show that most of this projection can be eliminated from further consideration. Let $G' = (V', E')$ where $v'_{k,\ell}$ corresponds to the k^{th} interval of u_3 and the ℓ^{th} interval of u_4. There is an edge in E' from v'_{k_1,ℓ_1} to v'_{k_2,ℓ_2} iff there exist i and j such that $(v_{i,j,k_1,\ell_1}, v_{i,j,k_2,\ell_2}) \in E$. Clearly, G' over approximates reachability. Thus, if a vertex of G' has no incoming edges, then all of the corresponding vertices in G must have no incoming edges as well. Computing the edges in E' requires examining all of the edges of E, but subsequent operations on the graph G' are much faster than those on G.

To reduce the time required to find edges of E, we start with a small value of m and thus a coarse grid. Many large blocks can be eliminated from G' even with a coarse grid. We then double m (i.e. divide each vertex of G' into four) and recompute reachability using the finer grid for finding edges in E as well. In practice this adaptive griding approach eliminates blocks quickly while achieving enough precision to allow the rest of the verification to proceed without difficulties.

B. Escape from the Failure Set

At the end of establishing differential operation, there are a few cubes with self-loops – there is more than one such cube because of the over approximations described above. These cubes contain the point $\mathbf{x}_{fail}$. We now construct a larger cube that contains all of these and make a change of variables so that this cube is centered at the origin. We'll write $\mathbf{x}$ for vectors in the original coordinate system and $\mathbf{u}$ for vectors in the coordinates where the center of a cube with a self-loop is at the origin. Let r be the maximum ℓ_2 distance of any point in this cube from the origin.

As described at the beginning of this section, we use piecewise quadratic models for transistor currents and model node capacitances as constants. Thus, the derivative function, f, is piecewise quadratic. Our repeated subdivision of cubes when establishing differential operation ensures that the cube containing $\mathbf{x}_{fail}$ is modeled by a simple quadratic (i.e. a single "piece"). We can write this model as:

$$\dot{\mathbf{u}} \;=\; A_0 + A_1\mathbf{u} + \textstyle\sum_{j=1}^{d}(\mathbf{u}^T A_{2,j}\mathbf{u})\mathbf{b}_j \tag{5}$$

where $\mathbf{b}_j$ is a unit vector corresponding to the j^{th} component of $\mathbf{u}$. We will assume wlog that the $A_{2,j}$ matrices are symmetric throughout paper.

To establish the hypotheses of Theorem 2, we again exploit the differential operation of the oscillator and choose $H = \mathrm{diag}([+1,+1,-1,-1])$. The two $+1$ elements of H anticipate a growing, differential component of the state, and the two -1 elements are for a diminishing common-mode component. Consider $(\mathbf{u}_2-\mathbf{u}_1)^T H(f(\mathbf{u}_2)-f(\mathbf{u}_1))$:

$$\begin{array}{rl} & (\mathbf{u}_2-\mathbf{u}_1)^T H(f(\mathbf{u}_2)-f(\mathbf{u}_1)) \\ = & (\mathbf{u}_2-\mathbf{u}_1)^T HA_1(\mathbf{u}_2-\mathbf{u}_1) \\ & +\,(\mathbf{u}_2-\mathbf{u}_1)^T H\sum_{j=1}^{d}((\mathbf{u}_2-\mathbf{u}_1)^T A_{2,j}(\mathbf{u}_2+\mathbf{u}_1))\mathbf{b}_j \end{array} \tag{6}$$

We now derive a lower bound for

$$\frac{(\mathbf{u}_2-\mathbf{u}_1)^T HA_1(\mathbf{u}_2-\mathbf{u}_1)}{(\mathbf{u}_2-\mathbf{u}_1)^T H(\mathbf{u}_2-\mathbf{u}_1)} \tag{7}$$

and an upper bound for

$$\left|\frac{(\mathbf{u}_2-\mathbf{u}_1)^T H\sum_{j=1}^{d}((\mathbf{u}_2-\mathbf{u}_1)^T A_{2,j}(\mathbf{u}_2+\mathbf{u}_1))\mathbf{b}_j}{(\mathbf{u}_2-\mathbf{u}_1)^T H(\mathbf{u}_2-\mathbf{u}_1)}\right| \tag{8}$$

when $(\mathbf{u}_2-\mathbf{u}_1)^T H(\mathbf{u}_2-\mathbf{u}_1) > 0$.

Equation 7 is a convex conic program and can be solved by standard techniques (see [20, chap. 4.4]); let $\mathsf{lin}_{\min}$ be the minimum value for Equation 7. To bound the magnitude of the quadratic term, let $\sigma_{\max}$ denote the largest singular value of any of the $A_{2,j}$ matrices. Then, for all $j \in 1\ldots d$,

$$(\mathbf{u}_2-\mathbf{u}_1)^T A_{2,j}(\mathbf{u}_2+\mathbf{u}_1) \;\le\; \sigma_{\max}(\mathbf{u}_2-\mathbf{u}_1)^T(\mathbf{u}_2+\mathbf{u}_1)$$

Therefore,

$$\begin{array}{rl} & \left\|\sum_{j=1}^{d}((\mathbf{u}_2-\mathbf{u}_1)^T A_{2,j}(\mathbf{u}_2+\mathbf{u}_1))\mathbf{b}_j\right\| \\ \le & \sqrt{d}\,\sigma_{\max}(\mathbf{u}_2-\mathbf{u}_1)^T(\mathbf{u}_2+\mathbf{u}_1) \end{array}$$

Noting that the largest singular value of H is 1, and $\|\mathbf{u}_2+\mathbf{u}_1\| \le 2r$, we get:

$$\begin{array}{rl} & (\mathbf{u}_2-\mathbf{u}_1)^T H\sum_{j=1}^{d}((\mathbf{u}_2-\mathbf{u}_1)^T A_{2,j}(\mathbf{u}_2+\mathbf{u}_1))\mathbf{b}_j \\ \le & 2r\sqrt{d}\,\sigma_{\max}\|\mathbf{u}_2-\mathbf{u}_1\|^2 \end{array} \tag{9}$$

By our choice of H,

$$(\mathbf{u}_2-\mathbf{u}_1)^T H(\mathbf{u}_2-\mathbf{u}_1) \;\le\; \|\mathbf{u}_2-\mathbf{u}_1\|^2 \tag{10}$$

Now, let $k = \mathsf{lin}_{\min} - 2r\sqrt{d}\sigma_{\max}$. Combining the results from Equations 6 through 10, we get

$$(\mathbf{u}_2-\mathbf{u}_1)^T H(f(\mathbf{u}_2)-f(\mathbf{u}_1)) \;\ge\; k(\mathbf{u}_2-\mathbf{u}_1)^T H(\mathbf{u}_2-\mathbf{u}_1)$$

If $k > 0$, then we can satisfy the conditions of Theorem 2. In practice, the conditions of Theorem 2 can be satisfied by choosing r to be sufficiently small.

C. Proper Oscillation

As described in Section III-C, we reduce the state space from four dimensions to two by replacing the differential equation model for the circuit with a differential inclusion. The space to be considered forms a ring: the outer boundary is determined by the assumption that all signals have voltages between ground and V_{dd}, and the inner boundary is established by eliminating trajectories in a neighborhood near x_{fail}. Figure 3 shows the remaining region. We use a collection of "spokes" as shown in Figure 4, and show that all trajectories in these wedges converge to a unique, periodic attractor. The computation has three parts:

1) Starting from each "spoke", show that all trajectories starting at that spoke eventually cross the next spoke.
2) Show that all trajectories starting from the inner or outer boundary eventually cross the next spoke.
3) Starting from one spoke, compute the reachable set until it converges to a limit set.

V. Results

We generated transistor models using HSPICE to determine drain-to-source currents for 0.18μ long and 1μ wide nMOS and pMOS devices with the gate and drain voltages swept from 0 to 1.8V in 0.01V steps. For the nMOS transistors, we assume that the source and body are at 0V, and for the pMOS devices, we assume that they are at 1.8V. We assume that all transistors have a length of 0.18μ, and obtain current for other widths by linear scaling from the 1μ data. For all inverters, we use pMOS devices that are twice as wide as the nMOS devices. All forward inverters have transistors of the same size, and likewise for the cross-coupled inverters. In the following, s denotes ratio of the cross-coupled inverter size to the forward inverter size. This section first presents the verification of an oscillator with $s = 1$. Then, the oscillator is verified for $0.673 \leq s \leq 2.0$.

The verification routines were implemented using Matlab with Coho used for the final reachability computation. All times were obtained running on a dual Xeon E5520 (quad core) 2.27GHz machine with 32GB of memory. The computations described here are all performed using a single core.

A. Verification with equal-size inverters

The first phase of the verification establishes differential operation. Initially, the computation partitions the space for each of the u_i variables into 8 regions, creating a total of $8^4 = 4096$ cubes to explore. After eliminating cubes that have no incoming or self-circulating flows, the remaining cubes are subdivided and rechecked until there are 64 intervals for each variable. Figure 2 shows the remaining cubes projected onto the common-mode variables, u_3 and u_4 at the end of this phase.

With 8 intervals per region, there are 752 cubes under consideration (18% of the total space). With each subdivision, the number of cubes remaining increases by a factor of roughly 4.6, and thus the volume of the space under consideration drops by about a factor of roughly 0.29. With 64 intervals per region, 74676 cubes remain (0.45% of the total space).

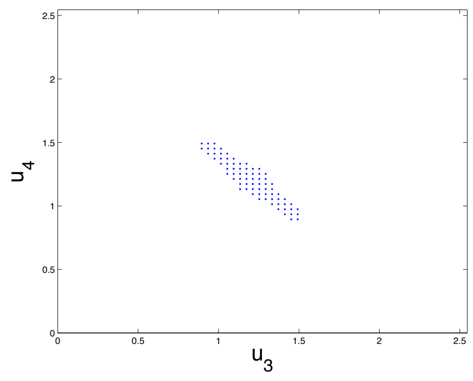


Fig. 2. Common-mode convergence to $V_{dd}\sqrt{2}/2$

The decrease in the volume is steady, suggesting that further reductions would be possible with more iterations. However, the time per iteration increases with the number of cubes under consideration, and the time for this phase dominates the total verification time. Thus, for verifying this circuit, there is no incentive to further refine the region bounding the common-mode signal.

The second phase of the verification eliminates the unstable equilibrium. The equilibrium is near the point where all node voltages are 0.867V. We chose U to be the hyper-rectangle with sides of length 0.1V whose center is at this point. The region U contains all cubes that correspond to graph-vertices with self-loops from phase 1. There is more than one such cube due to the use of interval arithmetic in computing the adjacency graph to ensure soundness. Using the least-squares best-fit quadratic model for points in U yields:

$$\begin{aligned} lin_{\min} &> 5 \times 10^{10}\text{sec}^{-1}, \\ \sigma_{\max} &< 2 \times 10^{9}\text{sec}^{-1}\text{V}^{-1}, \text{ and} \\ r &= 0.1\text{V} \end{aligned}$$

from which we get that the conditions of Theorem 2 are satisfied for any k with $0 < k < 4.92 \times 10^{10}\text{sec}^{-1}$. Thus, we can safely remove the cubes in U.

We can now repeat the procedure from phase 1 to remove all cubes that transitively have no incoming flows. This phase eliminates roughly half of the remaining cubes, leaving 38384 cubes for analysis by the final phase.

The final phase starts with the 38384 cubes from the second phase. As described in Section IV-C, we divide these cubes into 16 wedges divided by "spokes" in the $u_1 \times u_2$ projection. As described in Section IV-C, it is sufficient to show trajectories starting on the boundary of the wedge lead to points inside the next wedge in the clockwise direction. With 16 wedges, we perform 48 reachability computation runs. At this point, the oscillator is verified.

We also ran a longer reachability computation starting from a spoke and completing two complete cycles of the oscillation. The second cycle starts from a smaller region that the first and establishes tighter bounds on the limit cycle. The blue polygons in Figure 4 indicate this limit cycle. The remaining

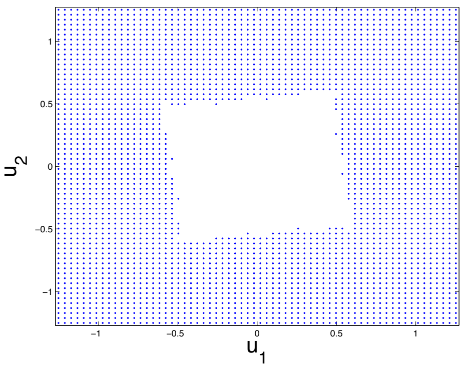


Fig. 3. Eliminating the unstable equilibrium

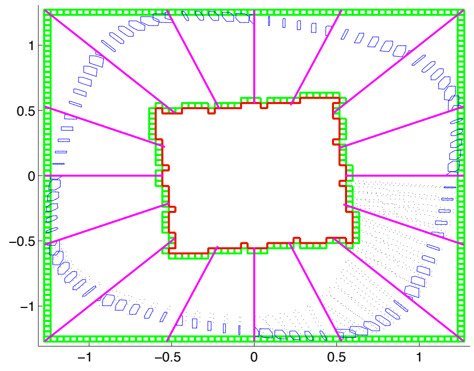


Fig. 4. Computing the invariant set

width of the limit cycle is mainly due to approximating the four-dimensional differential equation with a differential inclusion.

The computation is very efficient. The run-time of the first phase is about 720 seconds, and the run-time of reachability computation is less than 470 seconds. Reducing state space helps to improve performance significantly. It takes several hours to completes the reachability computation in full-dimensional space from a single cube. In contrast, the run-time in reduced-space is less than 10 minutes as shown above. Reducing the space also introduces over-approximations to the reachable regions. However, this did not lead to false-negative results, as the circuit converges to the oscillation orbit rapidly.

B. Verification for a range of sizes

Phases 1 and 3 of our verification method use conservative over-approximations to guarantee soundness of the results. These approximations make it straightforward to model s as being in an interval rather than having a precise value. We have verified escape from the failure set for values of s from 0.673 to 2.0 by testing values of s in steps of 0.01 for $0.6 \leq s \leq 2$ and in steps of 0.001 for $0.67 \leq s \leq 0.7$. The lower-bound for s is slightly higher than the one reported in [8]. We conjecture that our transistor current tables are slightly different than those used in [8] perhaps due to an updating of the SPICE models provided by the foundry. For $s > 2$, the third phase of the verification fails to show that trajectories leave the "corners" of the $u_1 \times u_2$ space. These correspond to lock-up of the cross-coupled inverters. The DC analysis method shows that these lock-up states become stable for $s > 2.25$. The gap between the reachability computation and the DC analysis is presumably due to conservative over-approximations used in the reachability method.

VI. Conclusions

This paper has presented the first, formal verification that the differential oscillator circuit presented in [2] properly starts from almost all initial conditions. In particular:

- no "physically plausible" oscillator starts from all initial conditions (Theorem 1, Section II);
- we presented a generalization of the "cone-argument" from [1] to show that the failures occur with probability zero and thus the oscillator starts with probability one (Theorem 2, Section III);
- our approach shows how reachability analysis can be combined effectively with dynamical systems analysis;
- we showed how differential-operation, a common feature of analog designs, can be exploited for model reduction.

We elaborate on some of these below.

First, metastable behaviors is unavoidable for most mode-switching circuits. While metastability is most often associated with synchronizer circuits [21], [22], it arises anytime the state of a continuous system can evolve to two or more distinct states. For example, when a phase-locked loop (PLL) locks, the VCO phase may advance to match the phase of the reference, or the VCO may drop back depending on the initial conditions. Thus, there are conditions where any physically realizable PLL takes an arbitrarily long time to lock. On the other hand, there are are published verifications of bounded lock time for phase-locked loops (e.g. [23]). The discrepancy is resolved by noting that [23] uses an abstract model for the phase-comparator that makes a discontinuous step as the phase-difference passes through 180°. For many designs, this is a reasonable abstraction; yet, we note that a PLL can fail to lock if there is a dead-spot in the response of the phase-comparator at the wrap-around point. We see our work as complementary to that of [23] – they provide powerful abstractions that enable the verification of larger designs, and we provide methods of ensuring that those abstractions are sound.

Second, our verification combined analytical methods from dynamical systems theory with reachability methods that are more typical of the formal methods community. Neither alone is sufficient to verify the oscillator. Reachability techniques are inadequate because they cannot show escape from a failure set of zero measure. Such "failures" are not of concern to practical designers as they are unobservable in the physical system. On the other hand, the dynamical systems methods that allow us to establish probability-one results are arguments about local dynamics. The reachability computations are needed to go from these local results to proving global properties.

The notion of probability that we used, a smooth distribution over initial states, was simplistic. A more physical model would use stochastic integration techniques to determine the evolution of this distribution under the circuit dynamics as perturbed by noise processes such as thermal noise. While this might be more satisfying, it would mainly serve to make the mathematics more complicated, and quantitative results would be hard to obtain due to the highly non-linear dynamics of the circuits. However, the basic topological observations on which we base our results would be preserved. Thus, we believe that our probability one results would continue to hold in a more detailed, stochastic model.

Proving that something happens "eventually" can be unsatisfying, as such proofs often don't give an indication of how long one needs to wait. Our proof for Theorem 2 shows that the divergence is at least as fast as an exponential with time-constant k. For the oscillator considered, $k \approx 1/(20\text{ps})$. Thus, we can make a quantitative conclusion that in a few nanoseconds, the probability that the oscillator has not started is extremely small. This should satisfy practicing designers.

Of course, there are many areas of future work. Most immediately, we claimed escape from the failure set for a wide-range of inverter sizes by verifying the property for a large number of closely spaced choices of the sizes. We would like to use interval-arithmetic methods to show that these intervals are completely covered. To do so, we are making a few extensions to the intlab package [24]. Likewise, we plan to show that the method can be applied to a design in a more state-of-the-art process (e.g. using PTM models [25]). We expect to include results for interval arithmetic and other processes in the final version of this paper.

We would like to verify larger circuits. For example, a ring oscillator with six or more stages can have stable higher harmonic modes if small inter-stage couplings are included in the model. We would like to verify (and refute) such designs. We expect that the first two phases of our verification could readily be generalized to a oscillators with an arbitrary number of stages with straightforward inductive formulations. We don't see induction working directly to extend the reachability analysis to larger designs. Instead, we are looking further into dynamical systems approaches to rule out entire classes of failure modes. Then we use reachability analysis techniques like those presented in the paper to complete the verification. The Rambus oscillator circuit is a good example for detailed analysis of how reachability computation complexity scales with circuit size.

References

[1] I. Mitchell and M. Greenstreet, "Proving Newtonian arbiters correct, almost surely," in *Proceedings of the Third Workshop on Designing Correct Circuits*, Båstad, Sweden, Sep. 1996.

[2] K. D. Jones, J. Kim, and V. Konrad, "Some "real world" problems in the analog and mixed-signal domains," in *Proc. Workshop on Designing Correct Circuits*, Apr. 2008.

[3] W. Hartong, L. Heidrich, and E. Barke, "Model checking algorithms for analog verification," in *Proceedings of the 39th ACM/IEEE Design Automation Conference*, Jun. 2002, pp. 542–547.

[4] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *Proceedings of 2004 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2004, pp. 210–217.

[5] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *Proceedings of Design Automation and Test Europe*, Mar. 2006, pp. 257–262.

[6] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda, "Verification of analog/mixed-signal circuits using labeled hybrid petri nets," in *Proceedings of the International Conference on Computer Aided Design*, Nov. 2006, pp. 275–282.

[7] J. Xiao, A. V. Peterchev, and S. R. Sanders, "Architecture and IC implementation of a digital VRM controller," in *IEEE* 32nd *Annual Power Electronics Specialists Conference (PESC'01)*, vol. 1, Jun. 2001, pp. 38–47.

[8] M. R. Greenstreet and S. Yang, "Verifying start-up conditions for a ring oscillator," in *Proceedings of the* 18th *Great Lakes Symposium on VLSI (GLSVLSI'08)*, May 2008, pp. 201–206.

[9] S. K. Tiwari, A. Gupta *et al.*, "fSpice: a boolean satisfiability based approach for formally verifying analog circuits," presented at the 2008 Workshop on Formal Verification for Analog Circuits (FAC'08), Princeton, NJ, Jul. 2008.

[10] S. Tiwari, A. Gupta *et al.*, "First steps towards SAT-based formal analog verification," in *Proceedings of the 2010 International Conference on Computer Aided Design*, Nov. 2010.

[11] S. Steinhorst, M. Peter, and L. Hedrich, "State space exploration of analog circuits by visualized multi-parallel particle simulation," in *International Conference on Signal Processing Systems (ICSPS'09)*, May 2009, pp. 858–862.

[12] M. H. Zaki, I. Mitchell, and M. R. Greenstreet, "Towards a formal analysis of DC equilibria of analog designs," Presented at the 2009 Workshop on Formal Verification for Analog Circuits (FAC'09), Grenoble, France, Jun. 2009.

[13] M. Fränzle, "HySAT: An efficient proof engine for bounded model checking of hybrid systems," *Formal Methods in System Design*, vol. 30, no. 3, pp. 179–198, 2007.

[14] S. Little and C. Myers, "Abstract modeling and simulation aided verification of analog/mixed-signal circuits," presented at the 2008 Workshop on Formal Verification for Analog Circuits (FAC'08), Princeton, NJ, Jul. 2008.

[15] C. Yan and M. R. Greenstreet, "Faster projection based methods for circuit-level verification," in *Proceedings of the 2008 Asia and South Pacific design automation conference (ASPDAC'08)*, Jan. 2008, pp. 410–415.

[16] M. W. Hirsch and S. Smale, *Differential Equations, Dynamical Systems, and Linear Algebra*. San Diego, CA: Academic Press, 1974.

[17] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control*. Springer-Verlag, 2005, pp. 258–273, LNCS 3414.

[18] D. Pollard, *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, 2001.

[19] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Algorithmic analysis of nonlinear hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 4, pp. 540–554, Apr. 1998.

[20] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[21] T. Chaney and C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 421–422, Apr. 1973.

[22] D. J. Kinniment, C. Dike *et al.*, "Measuring deep metastability and its effect on synchronizer performance," *IEEE Transactions on VLSI Systems*, vol. 15, pp. 1028–1039, Sep. 2007.

[23] 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.

[24] S. Rump, "INTLAB - INTerval LABoratory," in *Developments in Reliable Computing*, T. Csendes, Ed. Dordrecht: Kluwer Academic Publishers, 1999, pp. 77–104, http://www.ti3.tu-harburg.de/rump/.

[25] Y. Cao, "PTM: predictive technology model," http://ptm.asu.edu, 2008.

Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking

Yakir Vizel*, Orna Grumberg*, Sharon Shoham†
*Computer Science Department, The Technion, Haifa, Israel
†School of Computer Science, Academic College of Tel Aviv-Yaffo

***Abstract*—In this work we present a novel lazy abstraction-refinement technique for hardware model checking, integrated with the SAT-based algorithm IC3.**

In contrast to most SAT-based model checking algorithms, IC3 avoids unrolling of the transition relation. Instead, it applies local checks, while computing over-approximated sets of reachable states. We find IC3 most suitable for *lazy abstraction*, since each one of its local checks requires different information from the checked model.

Similarly to IC3, our algorithm obtains a series of over-approximated sets of states. However, when constructing the series, different abstractions are used for different sets.

If an abstract counterexample is obtained, we either find a corresponding concrete one, or apply refinement to eliminate *all* counterexamples of the same length. Refinement makes the abstractions more precise *as* needed, and *where* needed. After refinement, the computation resumes from the same step where it was interrupted. The result is an *incremental* abstraction-refinement algorithm where the abstraction is *lazy*.

We implemented our algorithm, called L-IC3, and compared it with the original IC3 on large industrial hardware designs. We obtained significant speedups of up to two orders of magnitude.

I. Introduction

In this work we introduce a novel lazy abstraction-refinement technique for hardware model checking, integrated with the SAT-based algorithm IC3 [3].

Model checking [5] is an automatic procedure that determines whether a given system satisfies a specification. In spite of its great success in verifying hardware and software systems, the applicability of model checking is impeded by its high space and time requirements.

The introduction of SAT-based model checking algorithms [1], [15], [12], [16], [3] significantly increased the size of the verified systems. Still, the search for improved, more scalable methods is neverending.

Most SAT-based model checking algorithms are based on an unrolling of the model's transition relation in order to traverse its state space. In contrast, the recently introduced IC3 algorithm [3] avoids such unrolling. To verify a safety property, IC3 gradually builds a series of sets of states $F_0, \ldots, F_i, \ldots$, where F_i over-approximates the set of states reachable within i steps from the initial states. The computation moves back and forth along the F_i's and strengthens them by eliminating unreachable states. This is done via *local reachability checks* between consecutive sets F_i and F_{i+1}. IC3 either reaches a fixpoint, in which case all reachable states satisfy the desired property, or returns a counterexample.

Abstraction-refinement is a well known methodology for tackling the state-explosion problem. Abstraction hides model details that are not relevant for the checked property. The resulting abstract model is then smaller. *Lazy abstraction* [10], [13], developed for software model checking, is a specific type of abstraction that allows hiding different model details at different steps of the verification.

In this work we develop, for the first time, a *lazy* abstraction-refinement framework for hardware. We use the *visible variables abstraction* [11], which is particularly suitable for hardware. However, we use it in a lazy manner in the sense that different sets of visible variables are used in different iterations of the state-space traversal.

We find the IC3 algorithm most suitable for lazy abstraction since its state traversal is performed by means of local reachability checks, each involving only two consecutive sets. Thus, at each check a different set of variables is relevant.

Our model checking algorithm, called L-IC3, thus integrates a lazy abstraction-refinement mechanism into IC3. Similarly to IC3, L-IC3 computes a series of over-approximating sets F_i, each referring to a certain time frame. However, L-IC3 considers abstractions of the model during this computation. When constructing F_{i+1}, we determine a set of variables U_i, needed for its construction, and abstract both *states and transitions* accordingly. The variables in U_i are referred to as "visible", while the others are invisible and treated as inputs.

The key ingredients of L-IC3 are therefore a series Ω of over-approximating sets of states F_i and an abstraction sequence $\bar{U}$ of sets of variables U_i.

L-IC3 works in stages. Each stage consists of an *abstract model checking* step, followed by a *refinement* step. At a given stage, the abstract model checking extends both Ω and $\bar{U}$ and checks if they include a potential abstract counterexample. If not, the sequences are further extended. If a potential abstract counterexample is found, the algorithm strengthens the sets F_i by eliminating abstract states that might be a part of an abstract counterexample.

We use a nonstandard notion of abstract counterexample, based on both Ω and $\bar{U}$. It consists of a sequence of abstract states connected by abstract transitions, satisfying: (i) each transition is based on a different abstraction U_i, and (ii) each abstract state intersects the set F_i at the corresponding time frame. Our notion of counterexample reflects the incorporation of lazy abstraction into the mechanism of computing Ω.

If an abstract counterexample is found, meaning that no strengthening is possible anymore based on the abstractions, the refinement step is invoked. Refinement applies just one iteration of a *concrete* variation of IC3, on the Ω computed by the abstract model checking. By doing so, it either finds a concrete counterexample or strengthens the F_i's so that *all* concrete counterexamples of length k are eliminated. In the latter case, the U_i's are also refined by adding more visible variables to each of them, *as* needed and *where* needed. Once refinement is finished we move to the next L-IC3 stage and the abstract model checking is re-invoked, continuing the computation from iteration k+1, with the refined sequences. This makes L-IC3 *incremental.*

L-IC3 terminates with either a fixpoint, in which case we conclude that the system satisfies the property, or with a concrete counterexample.

In summary, the main contribution of our work is a novel *lazy* abstraction-refinement technique for hardware. To the best of our knowledge this is the first time lazy abstraction is considered in the context of hardware. Our abstract model checking and refinement are SAT-based. Both avoid unrolling of the transition relation. Since our framework is based on a subtle combination of the abstract and concrete models, we provide theoretical arguments to its correctness.

In order to evaluate our new algorithm we compared it with IC3 on a set of large industrial designs and properties. We obtained speedups of up to two orders of magnitude. Our experiments demonstrate that our lazy abstraction indeed uses different sets of variables in different time frames. Moreover, only a small portion of the design's variables are used.

A. Related Work

[6] and [2] suggest optimizations and extensions to IC3, but they do not combine it with a lazy abstraction-refinement mechanism ([6] suggests the use of abstraction for IC3 but without implementation details nor results). In [14], [9], [7], [4], [8], SAT-based refinement is introduced. However, they use an unrolling of the model while we use local checks a-la IC3. Similarly to [14], [4], we also exploit an unSAT-core for refinement. However, we never unroll the model, while [14] does. Further, [14] is not incremental since after refinement it resumes its (abstract) model checking from time frame 0.

IC3 [3] is sometimes also viewed as an abstraction-refinement algorithm, since it refers to over-approximated sets F_i and the strengthening of these sets resembles refinement. However, the underlying model used by IC3 is concrete, and only the concrete transition relation is considered. We, on the other hand, alternate between abstract transition relations (in the abstract model checking step) and the concrete transition relation (in the refinement step). Our algorithm thus adds a layer of abstraction-refinement on top of this over-approximation-strengthening mechanism.

II. Preliminaries

Definition 1. A *finite state transition model* is a tuple $M = (V, U, \mathit{INIT}, \mathit{TR})$ where V is a set of variables, $U \subseteq V$ is a set of state variables, $V \setminus U$ is a set of input variables, $\mathit{INIT}(V)$ is a propositional formula over V describing the initial states and $\mathit{TR}(V, V')$ is a propositional formula over V and the next-state variables $V' = \{v' \mid v \in V\}$ describing the transition relation.

We assume that $\mathit{TR}(V, V') = \bigwedge_{v \in U} (v' = f_v(V, V'))$ where $f_v(V, V')$ is a propositional formula that assigns the next value to $v \in U$ based on current and next-state variables. Note that for an input variable $v \in V \setminus U$, f_v is not defined. From this point on M is a finite state transition model.

The set of boolean variables of M induces a set of states $S = \{0,1\}^{|V|}$, where each state $s \in S$ is given by a valuation of the variables in V. A formula over V (resp. V, V') represents the set of states (resp. pairs of states) obtained by its satisfying assignments. With abuse of notation we will refer to a formula η over V as a set of states and therefore use the notion $s \in \eta$ for states represented by η.

The formula $\eta[V \leftarrow V']$, or η' in short, is identical to η except that each variable $v \in V$ is replaced with v'.

For a formula η over $V \cup V'$ we use $\mathrm{Vars}(\eta) \subseteq V \cup V'$ to denote all (current or next state) variables appearing in η.

Definition 2. An *over-approximated reachability sequence* (OARS) with respect to a model M and a property AGp, denoted $\Omega(M, p)$, is a sequence $\langle F_0, \ldots, F_k \rangle$ of propositional formulas over V such that the following holds:

- $F_0 = \mathit{INIT}$
- $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$
- $F_i \wedge \mathit{TR} \Rightarrow F'_{i+1}$ for $0 \leq i < k$
- $F_i \Rightarrow p$ for $0 \leq i \leq k$

The set of states represented by F_i over-approximates the states reachable from *INIT* in at most i steps. We refer to i as *time frame (or frame) i*. When M and p are clear from the context we omit them and write Ω.

Definition 3. Let Ω be an OARS. A formula η is *inductive* up to j, if $F_j \wedge \eta \wedge \mathit{TR} \Rightarrow \eta'$. η is an *invariant* up to level j if $F_i \Rightarrow \eta$ holds for each $i \leq j$.

Note that if η is inductive up to j then $F_i \wedge \eta \wedge \mathit{TR} \Rightarrow \eta'$ holds for each $i \leq j$. Due to the properties of an OARS, η is an invariant up to j iff it is inductive up to level $j - 1$, and in addition $F_0 \Rightarrow \eta$ (initialization).

A. SAT-based Reachability via IC3

IC3 [3] is a SAT-based model checking algorithm that, given a model M and a property AGp, computes increasingly long sequences $\Omega(M, p)$. The algorithm works iteratively, where at iteration k, the OARS of length k is extended to an OARS of length $k + 1$ by initializing the set F_{k+1} and possibly updating previous sets (with index $i \leq k{+}1$). The computation continues until either a counterexample is found or a fixpoint is reached (i.e. $F_{i+1} \Rightarrow F_i$ for some i).

One of the main features of IC3 is the fact that no unrolling of the transition relation is needed. We give a brief overview of how it operates. More details are given along the paper as needed. For the exact details we refer the reader to [3].

IC3 starts by checking if $\mathit{INIT} \wedge \neg p$ or $\mathit{INIT} \wedge \mathit{TR} \wedge \neg p'$ is satisfiable, in which case a counterexample of length zero or one is found and the algorithm terminates. If both are unsatisfiable, F_0 is initialized to *INIT* and F_1 is initialized to p. $\langle F_0, F_1 \rangle$ is an OARS (it satisfies the conditions in Def. 2).

IC3 extends and updates Ω, while strengthening the F_i's. The kth iteration starts from an OARS $\langle F_0, \ldots, F_k \rangle$. Then F_{k+1} is initialized to p. Clearly, $F_k \Rightarrow F_{k+1}$ and $F_{k+1} \Rightarrow p$ hold. Therefore, the purpose of strengthening is to ensure that $F_k \wedge \mathit{TR} \Rightarrow F'_{k+1}$. This is done by checking that $F_k \wedge \mathit{TR} \wedge \neg p'$ is unsatisfiable. If this formula is satisfiable then a state $s \in F_k$ is retrieved from the satisfying assignment. s is a bad state since it reaches $\neg p$ (and by that violates $F_k \wedge \mathit{TR} \Rightarrow F'_{k+1}$). At this point, either s is reachable from *INIT*, in which case a counterexample exists, or s is unreachable and needs to be removed from F_k. In order to determine if s is reachable, IC3 checks the formula: $F_{k-1} \wedge \mathit{TR} \wedge s'$. If this formula is unsatisfiable, then s can be removed from F_k (since the property $F_{k-1} \wedge \mathit{TR} \Rightarrow F'_k$ of an OARS holds without it as well), and the same process is repeated for other states in F_k that can reach $\neg p$ (if any). However, if $F_{k-1} \wedge \mathit{TR} \wedge s'$ is satisfiable, a predecessor $t \in F_{k-1}$ of s is extracted and handled similarly to s in order to determine if t (which is also a bad state) is reachable from *INIT* or not. IC3 therefore moves back and forth along the F_i's, while retrieving bad states b and checking their reachability from *INIT* via local reachability checks of the form $F_i \wedge \mathit{TR} \wedge b'$. During this process, the F_i's are strengthened by removing bad states that are not reachable[1]. If a state in $F_0 = \mathit{INIT}$ is reached during the backwards traversal, then a counterexample is obtained.

Definition 4. Satisfiability checks of the form $F_i \wedge \mathit{TR} \wedge \eta$ (where $\mathrm{Vars}(\eta) \subseteq V \cup V'$) are called *$i$-reachability checks.*

B. Abstraction

Throughout the paper we consider the "visible variables" abstraction [11]. Let $M_c = (V, U, \mathit{INIT}, \mathit{TR})$ be a model and let $U_i \subseteq U$ be a set of state-variables. We refer to U_i as the set of "visible variables".

Given U_i, we consider an abstract model $M_i = (V_i, U_i, \mathit{TR}_i)$ of M_c where $\mathit{TR}_i = \bigwedge_{v \in U_i} (v' = f_v(V, V'))$ is an abstract transition relation, and $V_i = \{v \in V \mid v \in \mathrm{Vars}(\mathit{TR}_i) \vee v' \in \mathrm{Vars}(\mathit{TR}_i)\} \subseteq V$. Note that the behavior of invisible state variables (in $U \setminus U_i$) is nondeterministic.

We do not introduce an abstraction of *INIT* as part of M_i since we always consider the concrete set of initial states. M_i is an *abstraction* of M_c, denoted $M_c \preceq M_i$, in the sense that both its set of states and its transition relation are abstractions of the concrete ones. M_i induces a set of abstract states S_i which includes all valuations to V_i. Specifically, each concrete state $s \in S$ is abstracted by the abstract state $s_i \in S_i$ that agrees with s on the assignment to the joint variables in V_i. In this case we write $s \preceq s_i$. We sometimes refer to s_i as the set of concrete states it abstracts: $\{s \in S | s \preceq s_i\}$.

[1]In fact, in order to remove a bad state b from F_i, IC3 finds a clause c that is an invariant up to i and implies $\neg b$, and adds c to F_i as a conjunct.

In addition, TR is abstracted by TR_i in the sense that $\mathit{TR} \Rightarrow \mathit{TR}_i$. Formally, the relation $\{(s, s_i) \mid s \preceq s_i\}$ is a simulation relation from M_c to M_i.

Given an OARS $\Omega(M_c, p) = \langle F_0, \ldots, F_k \rangle$ and an abstract model M_i, we say that a formula η is *inductive* up to level j w.r.t. M_i, if $F_j \wedge \eta \wedge \mathit{TR}_i \Rightarrow \eta'$.

Lemma 5. *Any formula inductive up to j w.r.t. M_i is also inductive up to j w.r.t. M_c.*

The lemma holds since $\mathit{TR} \Rightarrow \mathit{TR}_i$. When we do not explicitly mention a model, we refer to inductiveness w.r.t. M_c. The notion of an invariant always refers to M_c.

C. Lazy Abstraction

As mentioned above, *lazy abstraction* [10] allows to use different details of the model at different iterations of the state-space traversal. We adapt the notion of lazy abstraction to abstraction based on *visible variables* [11], and allow different variables to be visible at different time frames.

Definition 6. An *abstraction sequence* w.r.t. a model M_c is a sequence $\bar{U} = \langle U_0, \ldots, U_k \rangle$ where $U_i \subseteq U$ for $0 \leq i \leq k$, is a set of visible state-variables. $\bar{U}$ is *monotonic* if $U_i \subseteq U_{i+1}$ for each $0 \leq i < k$.

An abstraction sequence $\bar{U}$ represents different levels of abstraction of M_c. It induces a sequence of abstract models $\langle M_0, \ldots, M_k \rangle$ where M_i is defined as in Sec. II-B. If $\bar{U}$ is monotonic, the induced sequence of abstract models is also monotonic in the sense that $M_0 \succeq \ldots \succeq M_k \succeq M_c$.

Definition 7. Let $\bar{U} = \langle U_0, \ldots, U_k \rangle$ be a monotonic abstraction sequence and $\Omega(M_c, p) = \langle F_0, \ldots, F_k \rangle$ an OARS. A sequence $s_i, \ldots, s_j$ of abstract states where $0 \leq i < j \leq k+1$ is an *abstract path from i to j* if (i) for each $i \leq l < j$, $(s_l, s_{l+1}) \models \mathit{TR}_l$, and[2] (ii) for each $i \leq l \leq \min\{j, k\}$, $s_l \cap F_l \neq \emptyset$.

An abstract path $s_0, \ldots, s_j$ from 0 to j is an *abstract counterexample of length j* if $s_j \cap \neg p \neq \emptyset$.

Note that the definition above is not standard. It refers to different transition relations at different steps. Also, it requires the abstract states to be part of the corresponding F_i.

Definition 8. An abstraction sequence $\langle {U_0}^r, \ldots, {U_k}^r \rangle$ is a *refinement* of an abstraction sequence $\langle U_0, \ldots, U_k \rangle$ if $U_i \subseteq {U_i}^r$ for each i.

III. Lazy Abstraction and IC3

In this section we describe our proposed algorithm for lazy abstraction, called L-IC3. The key ingredients of L-IC3 are an *abstraction sequence* $\bar{U}$ that induces different abstractions at different time frames as well as an *OARS* Ω.

L-IC3 starts with an initialization step and then works in *stages* (Fig. 1). Its initialization (lines 2-5) is similar to the

[2]Requirement (ii) dismisses paths that are known to be spurious based on Ω. $\min\{j, k\}$ is used for the case where $j = k + 1$, in which nonempty intersection is required only up to k.

```
1: function L-IC3($p$)
2:     $\Omega = \langle \mathit{INIT}, p\rangle; \bar{U} = \langle \text{Vars}(p)\rangle$
3:     if INIT-IC3($\Omega, \bar{U}, p$) == cex then
4:         return cex
5:     end if
6:     while A-IC3($\Omega, \bar{U}$) == abs-cex do
7:         if REFINE($\Omega, \bar{U}$) == cex then
8:             return cex
9:         end if
10:    end while
11:    return fixpoint
12: end function
```

Fig. 1: L-IC3

```
13: function A-IC3($\Omega, \bar{U}$)
14:     $k = |\Omega| - 1$
15:     while $\Omega.fixpoint()$ == false do
16:         $U_k = U_{k-1}$
17:         $\bar{U}.add(U_k)$
18:         $F_{k+1} = p$
19:         $\Omega.add(F_{k+1})$
20:         result = STRENGTHEN($\Omega, \bar{U}, k$)
21:         if result == abs-cex then
22:             return abs-cex
23:         end if
24:         $k = k + 1$
25:     end while
26:     return fixpoint
27: end function
```

Fig. 2: A-IC3

initialization of IC3 with one exception. If no counterexample of length 0 or 1 exists, then in addition to initializing Ω to $\langle F_0 = \mathit{INIT}, F_1 = p\rangle$, it initializes $\bar{U}$ to $\langle U_0 = \text{Vars}(p)\rangle$. Clearly, after initialization, Ω is an OARS.

Each L-IC3 stage (lines 6-10) consists of an abstract model checking step and a refinement step, both performed by variations of IC3. $\bar{U}$ and Ω are updated in both steps.

The abstract model checking extends and updates the OARS Ω until either a fixpoint is reached, or an abstract counterexample is found (line 6). In the latter case, the counterexample is *abstract* since it is computed w.r.t. the abstract transitions. However, it is also restricted by Ω (see Def. 7). A refinement is then performed (line 7). If the refinement finds a concrete counterexample then it terminates. Otherwise it refines $\bar{U}$ and updates Ω into an OARS (of the same length).

A new L-IC3 stage (line 6) of abstraction-refinement then begins, invoking A-IC3 with the updated Ω and the refined $\bar{U}$.

An invocation of L-IC3 results in either a fixpoint (in which case the property is proved), or a concrete counterexample.

A. Abstract Model Checking via A-IC3

The abstract model checking algorithm, A-IC3 (Fig. 2), either finds an abstract counterexample (line 22), or reaches a fixpoint (line 26) by computing an OARS Ω.

Using different abstractions The computation of Ω is done using a variation of IC3 which considers a *sequence of abstract* models, induced by a monotonic abstraction sequence $\bar{U} = \langle U_0 \dots, U_k\rangle$. Both abstract transition relations and abstract states are used. Even though abstract models are used, the obtained OARS satisfies the requirements of Def. 2, which refer to the concrete transition relation *TR*. To emphasize this, we sometimes refer to the sequence as a *concrete* OARS.

Recall that IC3 performs i-reachability checks of the form $F_i \wedge \mathit{TR} \wedge \eta$. A-IC3 also performs these checks (within STRENGTHEN, line 20), but instead of using the concrete *TR* it uses the *abstract* TR_i. This means that when traversing the model's state space, A-IC3 uses different abstract transition relations at different time frames. Further, when $F_i \wedge \mathit{TR}_i \wedge \eta$ is satisfiable, A-IC3 retrieves an *abstract* state $s_a \in M_i$ from the satisfying assignment. This abstract state is either used to strengthen Ω, or it is part of an abstract counterexample.

Incrementality If A-IC3 finds a counterexample at iteration k it returns. After refinement (line 7) A-IC3 is re-invoked with an updated Ω that is an OARS of the same length. The computation of Ω resumes from iteration $k + 1$ (line 14)[3].

Iterations In iteration $k \geq 1$, the OARS $\langle F_0, \dots, F_k\rangle$ and the abstraction sequence $\langle U_0, \dots, U_{k-1}\rangle$ are extended by 1 and updated as follows (see Fig. 2).

1) Check if a fixpoint is reached. If not:
2) U_k is initialized to U_{k-1} and added to $\bar{U}$.
3) F_{k+1} is initialized to p and added to Ω.
4) The sets $F_0, \dots, F_{k+1}$ are strengthened iteratively until $\langle F_0, \dots, F_{k+1}\rangle$ becomes an OARS, or an abstract counterexample is found.

Below we describe items 2 and 4 in more detail.

(2) Extending* $\bar{\mathbf{U}}$*: U_k is initialized to U_{k-1} (line 16). This is aimed at immediately eliminating from TR_k spurious transitions that lead from states in $F_{k-1} \subseteq F_k$ to $\neg p$ and were already removed from TR_{k-1}. Note that this initialization does not imply that the U_i sets will always be equal, since refinement might change them in different ways.

(4) Iterative Strengthening of* $\mathbf{\Omega}$*: A-IC3 obtains an OARS of length $k + 1$ by strengthening the F_i's s.t. no abstract counterexample of length $k+1$ exists w.r.t. the OARS $\langle F_0, \dots, F_k\rangle$. This is a sufficient condition to ensure that Ω is an OARS. For this purpose, A-IC3 finds abstract states that might be a part of an abstract counterexample at a certain time frame, and attempts to block them by learning corresponding invariants. Recall that the abstract counterexamples we consider are restricted not only by the abstract transition relations, but also by the F_i sets (Def. 6). Technically, such states are described by abstract proof obligations (similarly to the notion of proof obligations used in IC3).

Definition 9. An *abstract proof obligation*, or an *obligation* in short, is a pair (s_a, n) consisting of a level $n \leq k$ and an abstract state s_a s.t. (1) s_a is a "bad state" that reaches $\neg p$ along some abstract path, (2) $\neg s_a$ is an invariant up until n, (3) $s_a \cap F_{n+1} \neq \emptyset$, and (4) F_n reaches s_a in one step of TR_n.

Thus $n + 1$ is the minimal level intersecting s_a, and n is the minimal level reaching s_a in one *abstract* step. Note that it is

[3]An abstract counterexample is found w.r.t. $\Omega = \langle F_0, \dots, F_{k+1}\rangle$ produced in iteration k, where $|\Omega| = k + 2$. When A-IC3 is re-invoked, k is set to $|\Omega| - 1 = k + 1$.

possible that F_n cannot reach s_a along the concrete transitions. A-IC3 maintains two sets of obligations - *may* and *must*.

Definition 10. An obligation (s_a, n) is a *must obligation* w.r.t. iteration k if s_a must be shown unreachable from F_n in one step w.r.t. TR_n, in order to ensure that no abstract counterexample of length $k+1$ exists. All other obligations are *may obligations* w.r.t. k.

If s_a can reach $\neg p$ via an abstract path from level $n+1$ to level $k+1$, then (s_a, n) is a must obligation: unless s_a is blocked from F_{n+1} (by removing from F_n all states that reach s_a in one step), an abstract counterexample of length $k+1$ would exist. The same violation may also be reached from s_a in later levels F_j, $n+1 < j \leq k+1$, in which case it will be a suffix of a longer abstract counterexample with a longer prefix up to s_a. Therefore, we may also want to block s_a in F_j, $n+1 < j \leq k+1$. However, since different abstract transition relations are considered at each level, it is also possible that the same path leading from s_a to $\neg p$ is not valid from level $j > n+1$ since, for example, $U_j \supset U_{n+1}$ and hence the first transition along the path does not satisfy TR_j. The attempt to block a state s_a that is known to reach a violation from level $n+1$ in levels greater than $n+1$ creates *may* obligations[4].

The may obligations are not *required* to be blocked, but blocking them can prevent A-IC3 from encountering the same obligations/states in future iterations. On the other hand, if we report an abstract counterexample based on a may obligation, it is possible that no real abstract counterexample exists, resulting in an unnecessary refinement step which can damage the efficiency of the algorithm. We therefore greedily try to handle may obligations and strengthen Ω accordingly, but refrain from reporting abstract counterexamples based on them. Note that if a may obligation is in fact a must w.r.t. some greater k, then it will reappear as a must obligation in the following iterations.

In order to handle an obligation (s_a, n) and show s_a to be unreachable from F_n in one step, A-IC3 attempts to strengthen F_n by extracting predecessors t_a of s_a that satisfy $F_n \wedge TR_n \wedge s'_a$, defining new proof obligations based on them, and handling these obligations (by the same procedure). If F_n is successfully strengthened s.t. $F_n \wedge TR_n \wedge s'_a$ becomes unsatisfiable, then $\neg s_a$ becomes an invariant up to $n+1$.

Adding Invariants If $\neg s_a$ is an invariant up to $n+1$, then a stronger invariant that blocks s_a up to F_{n+1} is learned based on the *abstract model* M_n. Namely, $\neg s_a$ is strengthened to some sub-clause[5] c s.t. $F_0 \Rightarrow c$ and $F_n \wedge c \wedge TR_n \Rightarrow c'$, i.e. c is inductive up to n w.r.t. M_n and hence, by Lemma 5, also w.r.t. M_c. Consequently, c is also an invariant up to $n+1$, but it is a stronger invariant than $\neg s_a$ (since $c \Rightarrow \neg s_a$). The clause c is added as a conjunct to $F_0, \ldots, F_{n+1}$ while maintaining

[4]IC3 does not make a distinction between may and must obligations and handles them all the same since in the concrete case, a longer counterexample is always a *valid* path (its suffix reaching a violation is always valid).

[5]A state s_a is represented by a conjunction of literals, which makes its negation $\neg s_a$ a clause (i.e., a disjunction of literals). A sub-clause of $\neg s_a$ consists of a subset of its literals.

```
28: function REFINE($\Omega$,$\bar{U}$)
29:     result = C-STRENGTHEN($\Omega$)
30:     if result == cex then
31:         return cex
32:     end if
33:     REFINEABSTRACTION($\Omega$,$\bar{U}$)
34:     return done
35: end function
```

Fig. 3: REFINE procedure of A-IC3

the properties of a (concrete) OARS[6].

Key procedures used by A-IC3 are described in Sec. III-D.

B. Refinement

If A-IC3 finds an abstract counterexample of length $k+1$, refinement is invoked by L-IC3 (line 7). Refinement either finds a concrete counterexample or eliminates *all* concrete spurious counterexamples of length $k+1$. In the latter case, refinement also refines $\bar{U}$ to ensure that no *abstract* counterexample of length $k+1$ exists. Both an updated OARS $\Omega^r = \langle F_0^r, \ldots, F_{k+1}^r \rangle$ and a refined monotonic abstraction sequence $\bar{U}^r = \langle U_0^r, \ldots, U_k^r \rangle$ are returned.

The REFINE procedure is described in Fig. 3. REFINE first invokes C-STRENGTHEN, the strengthening procedure of the concrete IC3, on the sequence $\langle F_0, \ldots, F_{k+1} \rangle$ (whose prefix up to F_k is an OARS) obtained from the abstract model checking. If a concrete counterexample is found the algorithm terminates (lines 29-32). Otherwise, no concrete counterexample of length $k+1$ exists. Moreover, the updated (strengthened) sets $F_0^r, \ldots, F_{k+1}^r$ comprise an OARS. It remains to refine the abstraction sequence $\bar{U}$ in order to eliminate all *abstract* counterexamples of length $k+1$ as well. Thus, REFINEABSTRACTION is invoked (line 33).

REFINEABSTRACTION*:* A-IC3 found an abstract counterexample since it failed to strengthen the F_i's. Meaning, the relevant i-reachability checks $F_i \wedge TR_i \wedge t'_a$ could not be made unsatisfiable when using TR_i. C-STRENGTHEN, on the other hand, succeeds to do so. Namely, for each i-satisfiability check $F_i \wedge TR_i \wedge t'_a$ of A-IC3 that was satisfiable, C-STRENGTHEN manages to make the corresponding check $F_i^r \wedge TR \wedge t'$ for each $t \preceq t_a$ unsatisfiable, either by strengthening F_i^r or simply since it considers TR. Moreover, once $F_i^r \wedge TR \wedge t'$ becomes unsatisfiable, C-STRENGTHEN derives from it a clause $c \Rightarrow \neg t$ s.t. $F_i^r \wedge c \wedge TR \Rightarrow c'$ holds. C-STRENGTHEN strengthens Ω^r by adding c (invariant) as a new clause in all sets up to F_{i+1}^r. We consider it a *learned clause* at level $i+1$. The purpose of REFINEABSTRACTION is to ensure that for a learned clause c at level $i+1$, $F_i^r \wedge c \wedge TR_i^r \Rightarrow c'$ (with TR_i^r instead of TR) also holds. Meaning, c is inductive up to i w.r.t. M_i^r.

Lemma 11. *Let c be a clause learned by* C-STRENGTHEN *at level $i+1$. If $F_i^r \wedge \mathrm{TR}_i^r \Rightarrow {F_{i+1}^r}'$ then $F_i^r \wedge c \wedge \mathrm{TR}_i^r \Rightarrow c'$.*

Based on the previous lemma, in order to ensure $F_i^r \wedge c \wedge TR_i^r \Rightarrow c'$, it suffices to ensure unsatisfiability of $F_i^r \wedge TR_i^r \wedge \neg {F_{i+1}^r}'$ for every level $i+1$ in which learned clauses exist.

[6]c is not necessarily inductive w.r.t. M_i where $i < n$ (in case $U_i \subset U_n$).

To ensure unsatisfiability of a formula $F_i^r \wedge TR_i^r \wedge \neg {F_{i+1}^r}'$, we consider the same formula over *TR*, which is clearly unsatisfiable. We derive from it an unSAT-core. The next-state variables that appear in the unSAT-core, denoted $NS(\text{unSatCore}) = \{v \in V \mid v' \in \text{Vars}(\text{UnSatCore})\}$, are added to U_i.

Lemma 12. *Let $F_i^r \wedge \text{TR} \wedge \eta'$ be an unsatisfiable formula and let UnSatCore be its unsat core. Let $U_i^r \supseteq NS(\mathit{UnSatCore})$. Then $F_i^r \wedge \text{TR}_i^r \wedge \eta'$ is unsatisfiable.*

Finally, we propagate variables that were added to U_i^r forward in order to obtain a monotonic abstraction sequence. Since we only add variables to U_i^r, i.e. make the transition relation TR_i^r more precise, then the corresponding formulas remain unsatisfiable.

C. Correctness Arguments

The OARS obtained by L-IC3 is concrete. Specifically, it does not necessarily satisfy $F_i \wedge TR_i \Rightarrow F_{i+1}$. This results both from refinement that adds invariants learned based on the concrete *TR*, and from A-IC3 that learns an invariant based on some TR_i, but also adds it to F_{j+1} for $j < i$ even if it is not inductive w.r.t. TR_j. This complicates the correctness proof.

In particular, in IC3, when a proof obligation (s, n) is handled, then for any predecessor t of s, $\neg t$ is an invariant up to $n-1$, otherwise s would belong to a lower frame (since $F_i \wedge TR \Rightarrow F_{i+1}$). Now consider an abstract proof obligation (s_a, n). If we assume to the contrary that the predecessor t_a intersect some F_i (for $i < n$) then we can still deduce that the transition $(t_a, s_a) \models TR_n$ also exists at a lower frame, i.e. $(t_a, s_a) \models TR_i$ for $i < n$. This is since $TR_n \Rightarrow TR_i$ (recall that the same does not necessarily hold for $i > n$). However, if $t_a \cap F_i \neq \emptyset$, we cannot immediately deduce that $s_a \cap F_{i+1} \neq \emptyset$ since $F_i \wedge TR_i \Rightarrow F_{i+1}$ might *not* hold. It turns out that this property does hold (see Lemma 15), but more complicated arguments are needed, based on the following:

Lemma 13. *Let $\Omega = \langle F_0, \ldots, F_{k+1} \rangle$ and $\bar{U} = \langle U_0, \ldots, U_k \rangle$ be the sequences obtained at the end of a refinement step or at the end of an iteration of A-IC3 in the case that no counterexample was found. Then*

1) *Ω is an OARS.*
2) *For every clause c that was added to some F_i in Ω there exists some $j \geq i-1$ s.t. c is inductive up to j w.r.t. M_j.*
3) *No abstract counterexample of length $k+1$ exists w.r.t. the prefix $\langle F_0, \ldots, F_k \rangle$ of Ω.*

Theorem 14. *L-IC3 either terminates with a fixpoint, in which case the property holds, or with a concrete counterexample.*

D. Detailed Description of Strengthening

We now describe the procedures used by A-IC3 in detail.

STRENGTHEN *(Fig. 4)*: STRENGTHEN starts by checking $F_k \wedge TR_k \wedge \neg p'$ (line 37). If it is unsatisfiable, then $F_k \wedge TR \wedge \neg p'$ is also unsatisfiable as well (since $TR \Rightarrow TR_k$). Thus Ω is already an OARS and no further strengthening is needed.

Assume $F_k \wedge TR_k \wedge \neg p'$ is satisfiable. An abstract state $s_a \in M_k$ that reaches $\neg p$ in one abstract step is extracted

```
36: function STRENGTHEN($\Omega$,$\bar{U}$,$k$)
37:     while $F_k \wedge TR_k \wedge \neg p' == SAT$ do
38:         $obligations = \emptyset$
39:         retrieve abstract predecessor $s_k$
40:         if BLOCKSTATE($\Omega$,$s_k$,$k$,$k$,must) == abs-cex then
41:             return abs-cex
42:         end if
43:         while $obligations \neq \emptyset$ do
44:             $((s_a, n), handleMay) = $ CHOOSENEXT($obligations$)
45:             if $F_n \wedge TR_n \wedge s_a'$ == SAT then
46:                 retrieve abstract predecessor $t_n$
47:                 if BLOCKSTATE($\Omega$,$t_n$,$n$,$k$,must) == abs-cex then
48:                     if $handleMay$ then
49:                         $obligations.clearAllMust()$
50:                     else
51:                         return abs-cex
52:                     end if
53:                 end if
54:             else
55:                 $obligations.removeMust(s_a,n)$
56:                 BLOCKSTATE($\Omega$,$s_a$,$n+2$,$k$,may)
57:             end if
58:         end while
59:     end while
60:     PROPAGATECLAUSES($\Omega$)
61:     return done
62: end function
```

Fig. 4: Iterative strengthening of A-IC3

from the satisfying assignment, meaning $s_a \cap F_k \neq \emptyset$. All concrete states in $s_a \cap F_k$ can reach $\neg p$ via TR_k and therefore, if the property is to be proven, s_a must be blocked in F_k. Otherwise, an abstract counterexample exists.

In order to block s_a in F_k, STRENGTHEN calls BLOCKSTATE on the bad state s_a at level k (line 40). BLOCKSTATE either finds a counterexample or initializes the set(s) of obligations to reflect the need to block s_a (and possibly adds invariants to the F_i's).

STRENGTHEN then handles the proof obligations one at a time. CHOOSENEXT (line 44) first considers obligations from the must set only. Obligations are chosen in increasing order of their time frames. If the must set becomes empty, then as long as the may set is not empty, one may obligation with a minimal time frame is moved from the may set to the must set. STRENGTHEN then continues, with the exception that counterexamples are no longer reported.

Given a proof obligation (s_a, n):

- If F_n can indeed reach s_a in one (abstract) step, i.e., $F_n \wedge TR_n \wedge s_a'$ is satisfiable, then a predecessor t_a of s_a s.t. $t_a \cap F_n \neq \emptyset$ is extracted from the satisfying assignment (line 46). By Lemma 15, $t_a \cap F_i = \emptyset$ for all $i < n$. Thus $\neg t_a$ is an invariant up to $n-1$. Next, the state t_a needs to be blocked (eliminated) from level $l = n$ (line 47).
- When $F_n \wedge TR_n \wedge s_a'$ becomes unsatisfiable, the proof obligation (s_a, n) is removed (line 55) since s_a can no longer be reached from level n. In fact, $\neg s_a$ is now an invariant up to level $n+1$. In order not to encounter s_a in later iterations, we speculatively attempt to block (eliminate) s_a from level $l = n+2$, while using the may parameter (line 56).

A counterexample found by BLOCKSTATE is reported iff may obligations are not yet handled (lines 41 and 51).

```
63: function BlockState($\Omega$,$t_a$,$l$,$k$,$type$)
64:     if $l > k + 1$ then
65:         $min = k + 1$
66:     else
67:         $min$ = FindNonInductive($\Omega$,$\neg t_a$,$l - 1$,$k$)
68:         if $min$ == 0 then
69:             return abs-cex
70:         end if
71:         if $min \leq k$ then
72:             if type == must && min == l-1 then
73:                 $obligations.addMust(t_a, min)$
74:             else
75:                 $obligations.addMay(t_a, min)$
76:             end if
77:         end if
78:     end if
79:     AddInvariant($\Omega$,$\neg t_a$,$min$)
80:     return done
81: end function
```

Fig. 5: BLOCKSTATE procedure of A-IC3

Lemma 15. *Let (s_a, n) be a proof obligation, and let t_a be an abstract state such that $(t_a, s_a) \models \mathrm{TR}_n$. Then $t_a \cap F_i = \emptyset$ for every $i \leq n - 1$.*

BLOCKSTATE ***(Fig. 5):*** BLOCKSTATE(Ω,t_a,l,k,$type$) is used for blocking a "bad state" t_a from level l up to $k+1$, where $\neg t_a$ is already known to be an invariant up to $l - 1$.

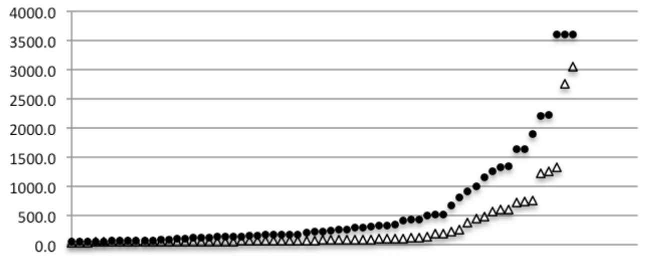
Note that if $l > k + 1$ (line 65) then t_a is already blocked up to $k + 1$. Thus $\neg t_a$ is added as an invariant up to $k + 1$ (line 79). Otherwise, BLOCKSTATE looks for a level such that $\neg t_a$ is invariant up to it.

Specifically, BLOCKSTATE looks for the minimal level min between $l-1$ and k s.t. $F_{min} \wedge TR_{min} \wedge t'_a$ is satisfiable (line 67). The important property is that $\neg t_a$ is an invariant up to min: If $min = l - 1$, this holds since $\neg t_a$ is already known to be an invariant up to level $l-1$ (this is also why the search for min starts at $l - 1$). If $min > l - 1$, then the fact that $F_{min-1} \wedge TR_{min-1} \wedge t'_a$ is unsatisfiable implies that $\neg t_a$ is inductive at $min - 1$ w.r.t. M_{min-1}, and hence, by Lemma 5 also w.r.t. M_c. Thus, it is an invariant up to min.

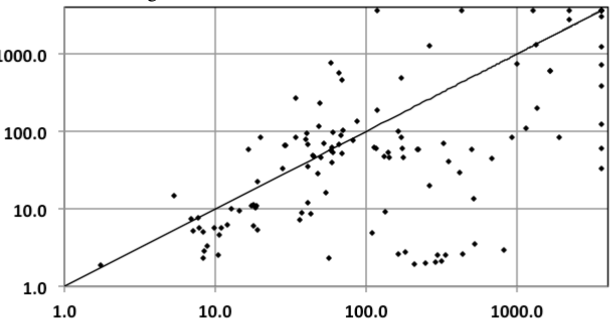
If $min = 0$, then the "bad state" t_a is reachable from *INIT* in one step of TR_0. Thus, an abstract counterexample is reported (line 69). If $min = k + 1$ then no corresponding level was found up to k, i.e., $\neg t_a$ is an invariant up to $k+1$ and no new proof obligation is added. However, if $min \leq k$ is found then the pair (t, min) is added as a new proof obligation (lines 72-76). Either way, $\neg t_a$ is added as an invariant up to min by calling ADDINVARIANT (line 79). ADDINVARIANT learns an invariant that strengthens $\neg t_a$ and adds it to $F_0, \ldots, F_{min}$.

Classifying obligations as may/must is performed in lines 72- 76 of BLOCKSTATE. Note that only obligations of the form $(t_a, l - 1)$ are must obligations.

PROPAGATECLAUSES*:* Similarly to IC3, if the main loop in STRENGTHEN terminates, added clauses are propagated forward by PROPAGATECLAUSES (line 60). Specifically, if $F_i \wedge c \wedge TR_i \wedge \neg c'$ is unsatisfiable then the clause c from F_i can safely be added to F_{i+1} while maintaining the properties of an OARS. This is done in order to get to a fixpoint.



(a) Runtime trend. Dots represent IC3, triangles represent L-IC3. Test-cases are sorted in an increasing runtime order.



(b) Comparing runtime. IC3 on X-axis and L-IC3 on Y-axis

Fig. 6: Runtime information for L-IC3 and IC3

IV. Experimental Results

For the implementation of the two algorithms we collaborated with *Jasper Design Automation*[7]. We used Jasper's formal verification platform in order to implement both the original IC3 and our L-IC3 algorithm. In both implementations we used optimizations from [6] (such as ternary simulation). Implementing these algorithms using Jasper's platform allowed us to develop and experiment with various real-life industrial designs and properties from various major semi-conductor companies. All designs contain thousands of state variables in the cone of influence of the properties.

The timeout was set to 3600 seconds and experiments were conducted on systems with Intel Xeon X5660 running at 2.8GHz and 24GB of main memory.

We experimented with 122 real safety properties from different designs. Fig. 6 shows two different analyses comparing the runtime of L-IC3 and IC3. Runtime trends are shown in Fig. 6a. As can be seen, the overall trend is in favor of L-IC3. In Fig. 6b runtime for IC3 and L-IC3 is represented by the X-axis and Y-axis respectively. We can clearly see the advantage of using L-IC3 on the more complicated test cases. These test cases are represented by the dots that are below the diagonal by a big margin. On these examples, the improvement in runtime is up to ***two*** orders of magnitude. The cases where IC3 performs better are usually cases where L-IC3 spends most of the time in refinement. Also, for false properties (counterexample exists), the performance of L-IC3 is affected by the way we treat *may* and *must* obligations. Due to our special handling, L-IC3 may lose the ability to

[7]An EDA company: http://www.jasper-da.com

N	♯Vars	Laziness - Time Frames and Number of Vars																	
		♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV	♯TF	♯AV
f_1	11866	[0-0]	323	[1-1]	647	[2-2]	686	[3-3]	699	[4-4]	705	[5-5]	713	[6-6]	714	[7-7]	728	[8-8]	743
		[9-9]	752	[10-10]	755	[11-11]	761	[12-12]	767	[13-13]	777	[14-14]	783	[15-15]	789	[16-18]	811		
f_2	5693	[0-7]	12																
f_3	5693	[0-0]	8	[1-1]	56	[2-2]	64	[3-3]	74	[4-4]	82	[5-7]	91						
f_4	5693	[0-6]	31	[7-7]	42	[8-8]	51	[9-13]	54										
f_5	5773	[0-0]	260	[1-1]	381	[2-2]	401	[3-3]	419	[4-34]	430								
f_6	1183	[0-0]	185	[1-1]	248	[2-2]	255	[3-3]	259	[4-4]	262	[5-5]	268	[6-8]	270	[9-9]	273	[10-30]	274
f_7	1247	[0-0]	57	[1-1]	62	[2-2]	73	[3-7]	76										
f_8	1247	[0-0]	63	[1-1]	64	[2-2]	72	[3-6]	83										
f_9	1277	[0-0]	263	[1-1]	303	[2-2]	318	[3-3]	321	[4-4]	322	[5-5]	323	[6-26]	347				
f_{10}	1389	[0-0]	253	[1-1]	304	[2-2]	324	[3-3]	341	[4-4]	351	[5-5]	355	[6-7]	363	[8-9]	399	[10-10]	409
		[11-12]	415	[13-13]	419	[14-16]	429	[17-18]	431										
f_{11}	1183	[0-0]	79	[1-1]	113	[2-9]	114												
f_{12}	1204	[0-0]	58	[1-1]	67	[2-2]	75	[3-7]	76										
f_{13}	3844	[0-0]	470	[1-1]	504	[2-2]	528	[3-3]	533	[4-4]	534	[5-11]	650						
f_{14}	3832	[0-0]	333	[1-1]	365	[2-2]	386	[3-5]	391	[6-6]	442	[7-10]	446						
f_{15}	3854	[0-0]	428	[1-1]	453	[2-2]	495	[3-3]	499	[4-4]	503	[5-5]	560	[6-6]	574	[7-7]	576	[8-10]	577
f_{16}	3848	[0-0]	432	[1-1]	462	[2-2]	487	[3-3]	498	[4-4]	501	[5-5]	634	[6-6]	650	[7-13]	658		
f_{17}	3854	[0-0]	426	[1-1]	480	[2-2]	525	[3-3]	539	[4-4]	540	[5-5]	559	[6-11]	570				
f_{18}	3848	[0-0]	469	[1-1]	547	[2-2]	551	[3-3]	553	[4-4]	635	[5-5]	672	[6-10]	674				

TABLE I: Lazy abstraction. N stands for the name of the verified property. ♯*Vars* stands for the number of state variables in the concrete model M_c. ♯TF stands for the time frames and ♯AV represents the number of variables (defining the abstract TR_i) in the abstract model M_i at the given time frame i (appearing in the column ♯TF).

find a counterexample which is longer than the length of the computed Ω. In those cases, IC3 may perform better. Note that the scatter at the middle is a bunch of comparable properties where both algorithms are on par.

In the given timeout, 7 properties cannot be solved by IC3 but are solved by L-IC3; 5 properties cannot be solved by L-IC3 but are solved by IC3. There are also 5 properties that cannot be solved by either algorithm. The overall runtime for IC3 is 75558 seconds while for L-IC3 it is 55424 seconds.

The laziness of our abstraction-refinement algorithm is demonstrated in Table I. The table shows how the abstraction is refined along increasing time frames. Different frames contain different variables that are needed in order to prove or disprove the given property. This demonstrates the fact that L-IC3 indeed takes advantage of the lazy abstraction framework.

Table II presents runtime characteristics for L-IC3 and IC3. In particular, it shows the number of clauses and the number of variables in Ω when either a fixpoint or a counterexample is found. In many of the examples the number of clauses produced by L-IC3 for its Ω is significantly smaller than the number of clauses produced by IC3. Recall that each of the clauses is learned via several local reachability checks. The reduced number of clauses thus indicates that L-IC3 applies a smaller number of checks and therefore issues a smaller number of calls to the SAT solver. This can explain the speedups it obtains.

An additional reason for the speedups is the fact that the local reachability checks of L-IC3 are easier than those of IC3. This is because the abstract transition relations TR_i are much smaller (in number of variables) than TR (see table I). Further, the sets F_i, computed by L-IC3 are smaller than those computed by IC3 (see Table II).

Recall that in Section III-A we distinguish between *must* and *may* obligations. The results reported above are obtained while using this distinction and handling all the *may* obligations after the *must* obligations, as described there. We also tried other configurations. For example, we ran experiments that do not distinguish between *must* and *may* obligations. Our experiments show that distinguishing between the two yields a better overall performance.

N	♯Vars	Stat	♯V[Ω]	♯V[Ω_L]	♯C[Ω]	♯C[Ω_L]	k	k_L	T	T$_L$
f_1	11866	false	1001	**818**	8457	**3939**	15	18	1646	**599**
f_2	5693	true	236	**11**	617	**62**	14	8	133	**9.2**
f_3	5693	true	229	**121**	1314	**570**	13	8	351	**40.5**
f_4	5693	true	104	**24**	2101	**32**	32	14	513	**13.6**
f_5	5773	true	$>$ 616*	**414**	$>$ 16689*	**12425**	7*	35	TO	1223
f_6	1183	true	432	**370**	50511	**29316**	36	31	2216	2763
f_7	1247	true	250	**152**	10732	**238**	11	8	432	**2.6**
f_8	1247	true	177	**96**	14702	**293**	8	7	520	**3.5**
f_9	1277	false	357	**331**	8762	**3788**	13	27	164	**101**
f_{10}	1389	false	**397**	417	**12455**	19742	13	19	**262**	1268
f_{11}	1183	true	114	**106**	29183	**2589**	9	10	1153	**109**
f_{12}	1204	true	114	**105**	18698	**229**	8	8	818	**3.0**
f_{13}	3844	true	**320**	578	**547**	1529	10	12	**16.7**	59.1
f_{14}	3832	true	650	**488**	2414	**1553**	12	11	117	**61**
f_{15}	3854	true	$>$ 470*	**666**	$>$ 8320*	**5363**	6*	11	TO	**730**
f_{16}	3848	true	$>$ 687*	**826**	$>$ 7733*	**5506**	8*	14	TO	**381**
f_{17}	3854	true	811	**673**	10934	**1837**	13	12	919	**83**
f_{18}	3848	true	898	**716**	9889	**2080**	13	11	1891	**84**
f_{19}	3848	true	**966**	$>$ 216*	**13370**	$>$ 266*	11	7*	**2225**	TO

TABLE II: Running parameters for various properties. *N* stands for the name of the verified property. ♯*Vars* stands for the number of state variables in the cone of influence. ♯V[Ω] - number of variables in Ω, ♯C[Ω] - number of clauses in Ω, k - size of $\Omega(M, p)$ and T - the runtime in seconds. The subscript L represents the value for the *Lazy* version (L-IC3).

In addition to the industrial experiments, we also ran experiments on the HWMCC'11 benchmark. We used the test-cases with single properties. Most of the properties in this benchmark are fairly easy and can be solved in a matter of a few seconds both by IC3 and L-IC3. There are also a few cases where IC3 performs better or even reaches a result while L-IC3 does not. In these cases L-IC3 spends most of the time in refinement. On the other hand, there are several test cases that can only be solved by L-IC3 while IC3 reaches timeout.

V. Acknowledgments

The authors would like to thank Håkan Hjort, Ziyad Hanna and Yael Meller for valuable comments. Jasper Design Automation is thanked for the help in conducting the experiments.

References

[1] A. Biere, A. Cimatt, E. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *TACAS*, 1999.
[2] A. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An incremental approach to model checking progress properties. 2011.
[3] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.
[4] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, 2005.
[5] E. Clarke and D. Peled O. Grumberg. *Model Checking*. MIT press, 1999.
[6] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
[7] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *ICCAD*, 2003.
[8] A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *CAV*, 2005.
[9] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
[10] T.A. Henzinger and R. Majumdar R. Jhala. Lazy abstraction. In *POPL*, 2002.
[11] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
[12] K. L. McMillan. Interpolation and SAT-based Model Checking. In *CAV*, 2003.
[13] K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.
[14] K. L. McMillan and N. Amla. Automatic Abstraction without Counterexamples. In *TACAS*, 2003.
[15] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, 2000.
[16] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *FMCAD*, 2009.

IC3-Guided Abstraction

Jason Baumgartner, Alexander Ivrii, Arie Matsliah, Hari Mony
IBM Corporation

***Abstract*—*Localization* is a powerful automated abstraction-refinement technique to reduce the complexity of property checking. This process is often guided by SAT-based bounded model checking, using counterexamples obtained on the abstract model, proofs obtained on the original model, or a combination of both to select irrelevant logic. In this paper, we propose the use of bounded invariants obtained during an incomplete IC3 run to derive higher-quality abstractions for complex problems. Experiments confirm that this approach yields significantly smaller abstractions in many cases, and that the resulting abstract models are often easier to verify.**

I. Introduction

Automated property checking techniques hold considerable promise to mitigate what has become one of the most important problems facing the semiconductor industry today: the *verification crisis*. Through the advent of numerous advanced proof, falsification, abstraction and reduction techniques, formal property checking has scaled to the necessary level to address many practical industrial applications, and has become an essential CAD technology. However, many problems remain beyond the capacity of current property checking algorithms, thus continued advances are of critical importance.

Localization is a powerful abstraction technique which reduces the size of a verification problem by replacing gates by *cutpoints*, which act as unconstrained nondeterministic variables. Because the cutpoints may simulate the behavior of the original gates, this approach over-approximates the behavior of a design hence is sound yet incomplete. *Refinement* is used to eliminate spurious failures on the abstract design by eliminating cutpoints which are deemed responsible for the failure. Ultimately, the abstract design is passed to a proof engine. It is desirable that the abstract design be as small as possible to enable more efficient proofs, while being immune to spurious counterexamples.

Various techniques have been proposed to guide the abstraction-refinement process of localization. Most state-of-the-art localization implementations use SAT-based bounded model checking to select the abstract netlist upon which an unbounded proof is attempted, given relative scalability of bounded model checking vs. proof techniques. This abstraction process is either based upon counterexamples obtained on the abstract design [1], based upon proofs obtained on the original design [2], or a hybrid of both [3]. It has been noted that the latter approach tends to yield the smallest abstractions, albeit at the cost of additional runtime. Specifically, the abstraction-refinement process relies upon heuristics and thus may include unnecessary logic, in turn entailing unnecessary proof complexity. In hybrid approaches, one approach is often used to re-process the abstraction computed by the other, in an attempt to eliminate this unnecessary logic. The additional runtime spent on the abstraction-refinement process is often a worthwhile strategy for overall minimal resources.

Various SAT-based unbounded proof techniques have been developed which in cases are very scalable, such as IC3 [4]. One powerful characteristic of IC3 is that when successful, it may yield a proof or counterexample while analyzing only a small approximation of the overall behavior of the design under verification. In the case of a proof, an often-compact inductive invariant is derived. In the case of a counterexample, a directed search from the initial states toward the property found a path, in cases requiring less effort than bounded model checking.

While often efficient, the intrinsic complexity of property checking explains why IC3 often fails to solve a complex problem given practical resources. Furthermore, while IC3 intrinsically attempts to analyze the design in an abstract manner, it is well-known that abstraction may synergistically boost the scalability of various verification algorithms, even approximate ones. In this paper, we seek to exploit this synergy in two ways. **(1)** We extract design insight from an incomplete IC3 run via its (bounded) invariants. **(2)** We use the extracted information to improve the quality of localization, and thereby boost the scalability of subsequent verification algorithms including IC3.

II. Preliminaries

We focus on verification of safety properties on finite state machines (FSMs). An FSM M is a tuple $\langle X, I, T \rangle$, where X is a set of Boolean state variables, such that each assignment $s \in \{0,1\}^X$ corresponds to a state of M, and the predicates $I \subseteq \{0,1\}^X$ and $T \subseteq \{0,1\}^X \times \{0,1\}^X$ define its initial states and the transition relation, respectively. A predicate $P \subseteq \{0,1\}^X$ defines a property to be verified on M.

State variables and their negations are called *literals*, and disjunctions (conjunctions) of literals are called *clauses* (*cubes*). A CNF formula is a conjunction of clauses. We follow the standard notation of X' representing next-state variables, and derive CNF formulas from FSMs in a straight-forward way.

A sequence π of states $t_0, \ldots, t_n$ is a *path* if for each $0 \leq i < n$, $\langle t_i, t_{i+1} \rangle \in T$. If t_0 is an initial state, the path is called *initialized.* A state t is *reachable* if there is an initialized path that ends in t. Let R denote the set of all reachable states, and for $k \geq 0$, let R_k denote the set of states reachable by initialized paths of length at most k. The verification objective is to prove $R \subseteq P$.

A CNF formula φ is an *invariant* if $s \in R \implies s \models \varphi$. Furthermore, φ is a *k-step invariant* if $s \in R_k \implies s \models \varphi$. A CNF formula φ is an *inductive invariant* if $I \implies \varphi$ and $\big((s \models \varphi) \wedge (\langle s, s' \rangle \in T)\big) \implies s' \models \varphi$.

If φ is an inductive invariant and $\varphi \implies P$, then $R \subseteq P$ and φ is called an *inductive proof of P*.

III. Generating Hints with IC3

In this section we briefly describe IC3 [4], [5]. This algorithm proceeds by incrementally refining and extending a sequence $\mathcal{F}_1, \ldots, \mathcal{F}_k$ of sets of clauses (termed *bounded invariants*) referring solely to state variables. For simplicity we define $\mathcal{F}_0 = I$. Throughout the run of IC3 the following invariants are preserved:

- $\mathcal{F}_i \Rightarrow \mathcal{F}_{i+1}$, for $0 \leq i \leq k-1$,
- $\mathcal{F}_i \supseteq \mathcal{F}_{i+1}$ as sets of clauses, for $1 \leq i \leq k-1$,
- $\mathcal{F}_i \wedge T \Rightarrow \mathcal{F}'_{i+1}$, for $0 \leq i \leq k-1$,
- $\mathcal{F}_k \Rightarrow P$.

Initially $k = 1$, and is increased whenever $\mathcal{F}_k \wedge T \Rightarrow P'$ holds. Note that each $\mathcal{F}_i$ constitutes an overapproximation of R_i, and $\mathcal{F}_1 (= \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k)$ implies that P holds in the first k timesteps. While processing bound k, the condition $\mathcal{F}_k \wedge T \Rightarrow P'$ might fail to hold, and then IC3 attempts to propagate additional information to sets $\mathcal{F}_{i \leq k}$ to address this potential failure. If it cannot, this signifies a true counterexample. Otherwise, each added clause is *pushed* to the highest-possible $\mathcal{F}_i$. Some bounded invariants may be determined to be unbounded invariants, and when that set implies P an inductive proof has been completed.

A. Hint Generation

Recall that the conjunction of clauses in $\mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k$ implies that P holds for the first k timesteps. This leads to the idea of creating a localization including only the state variables appearing in these clauses. However, as with traditional localization, some of these state variables may be unnecessary to complete a proof of correctness.

Note that there is additional information one may draw on the relevance of various state variables from the IC3 invariants which may be used to improve the abstraction quality. In particular, for each IC3 invariant c one can consider whether it is an unbounded invariant, the first bound k for which c was introduced, and the maximal frame i that it can be pushed to relative to k. For each state variable, one can analyze the number or the proportion of bounded invariants or clause sets it belongs to. In this section we address how to use the sets $\mathcal{F}_i$ to provide hints to localization on the relative *priority* of various state variables, represented by integers with the convention that a lower number reflects a higher priority.

We experimented with numerous heuristics for assigning priorities, and our best method (**PM1**) is as follows. The priority of each state variable is initially ∞, and may be decreased during the IC3 run. Whenever a new bounded invariant clause c is produced, all state variables referenced by c that have priority ∞ have their priorities updated to the current bound k being processed. I.e., after the run the priority of each state variable represents the earliest bound requiring a corresponding clause for a proof of validity of P, and the clauses for proofs of smaller bounds have higher priorities. Also note that an abstraction including the state variables with priorities $\leq i$ will satisfy P for the first i timesteps.

(PM2) In this variant, the priority of each state variable is initially ∞, and when a new bounded invariant clause c is produced that is "pushable" to frame $i \leq k$, we update the priority of each latch participating in c to $k - i$, unless a smaller number was assigned to it earlier. If c is an unbounded invariant, then the priorities of those latches in c are updated to 0.

(PM3) The priority of each state variable is initially ∞, and whenever a new invariant clause c is produced (bounded or unbounded), we update the priority of each latch participating in c to 0.

(PM2) and **(PM3)** performed generally worse than **(PM1)**, hence we restrict our experimental focus to **(PM1)**. Continued experiments with alternate heuristics is subject of ongoing research.

IV. Localization with IC3 Hints

In this section, we detail how the priorities assigned on the state variables may be used to guide localization. The localization abstraction is done by selection of state variables: if included, its entire next-state function will also be included, else the state variable will be replaced by a cutpoint. The localization is performed by interleaving counterexample based abstraction (CBA) and proof-based abstraction (PBA) in a bottom-up manner [6]. We start with no state variables included, and perform SAT-based bounded model checking for one timestep on the abstract design. Spurious counterexamples are analyzed and the abstraction is refined by adding state variables which are deemed adequate to rule out the spurious behavior. Once there are no more counterexamples of a given depth, we perform proof-based abstraction for that depth to attempt to eliminate unnecessary logic. We then increment the depth and repeat the process for a configurable resource limit.

The key challenge in localization abstraction is the refinement used to to eliminate the spurious counterexamples. Several approaches have been proposed to minimize the amount of logic added to refute a spurious counterexample, e.g., [7], [1], [8]. Our approach relies upon counterexample trace minimization to minimize the number of cutpoints assigned in a spurious counterexample, using a combination of SAT and ternary analysis [7], [6]. Once the counterexample has been minimized, all cutpoints assigned in that trace are refined.

Even with aggressive trace minimization, this minimization is heuristic and greedy, which may entail a suboptimal abstraction. Furthermore, minimality is not unique, and across multiple refinements it is likely that cumulative refinement choices result in unnecessary logic being included. This is the motivation for using PBA to attempt to further reduce the CBA. We propose to improve this process to a greater degree by using the IC3 priorities to guide the abstraction via one of the following refinement methods.

(RM1) Start CBA with empty abstraction. When refining the abstract model by adding state variables that are assigned in the spurious counterexample during CBA, we only add the subset of those with the highest IC3 priority.

(RM2) In addition to using **(RM1)**, we begin with the abstract model including all state variables of highest IC3 priority vs. starting with an empty abstraction.

The aim of **(RM1)** is to skew CBA to avoid including state variables with lower IC3-assigned priorities in the abstract model. Even though this may result in an increased number of refinements during CBA, our experiments demonstrate that this guidance results in smaller abstractions in practice. **(RM2)** was developed based on the observation that the abstract model often ultimately includes all state variables with the highest IC3-assigned priority regardless; beginning with this set reduces abstraction-refinement runtime due to fewer refinements, and fewer refinements entail fewer heuristic mistakes which bloat abstraction size. We have experimentally found that **(RM2)** yields smaller abstractions, hence we restrict our experimental focus to **(RM2)**.

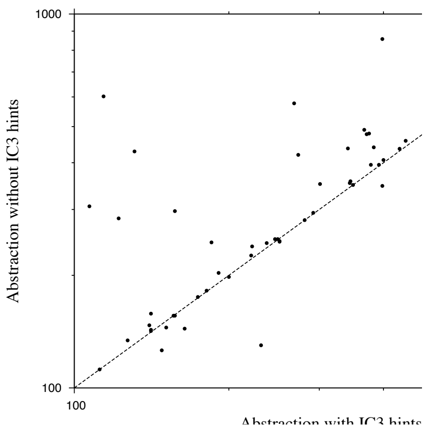


Fig. 1. Number of localized state variables with vs. without IC3 hints

V. Experiments

In this section we present our experiments. All experiments were performed on 2.0Ghz Linux-based machines with 4Gb of RAM, using the techniques presented in this paper as implemented in the IBM verification tool *SixthSense* [9]. We focused upon the single-target benchmarks from HWMCC 2011 [10].

A. *Effect on Abstraction Size*

This first set of experiments compares abstraction sizes generated with and without IC3 hints. Both include an interleaved CBA/PBA localization for a time limit of 300 seconds. For the IC3 hint-guided abstraction, we run IC3 for 120 seconds to generate the priorities, then proceed with the abstraction-refinement loop **(RM2)**. 294 instances that were solved by IC3 within the 120 seconds limit, or during localization, are excluded from the analysis. In the remaining instances, the total number of state variables in the abstracted models reduces from 47994 to 41036 when using IC3 hints: a cumulative reduction of 14.5%, with median reduction of 6.8%. Figure 1 depicts the reduction on instances with 100 to 1000 state variables after localization.

These results clearly show improved abstraction size in most cases using IC3 hints. There are some which yield worse abstractions, though we note that this is likely inevitable in rare cases given the heuristic nature of abstraction.

B. *Effect on IC3 Resources*

To demonstrate that the reduced abstract model size improves verification resources, we used IC3 with a 900 second timeout on the localized design. Of the 171 HWMCC 2011 benchmarks which are unsolved by the first 120-second IC3 or during localization, 15

Design	**Traditional Abstraction**	**IC3 Hint-Guided Abstraction**
6s9	TO	357s
6s19	TO	519s
6s43	TO	236s
6s50	TO	484s
6s51	TO	198s
bc57sensorsp0	TO	838s
pdtvsarmultip29	TO	473s
pdtswvtma6x6p2	TO	665s
pdtswvtma6x6p1	TO	333s

TABLE I
IMPACT OF HINT-GUIDED LOCALIZATION ON SUBSEQUENT IC3 PROOF RUNTIME

were solved by the heavier-weight IC3 using traditional abstraction-refinement without IC3 hints. In contrast, 24 were solved using abstraction with IC3 hints; a proper superset. While this is only a modest improvement in terms of conclusive solutions, we note that **(1)** 900 second IC3 runtime is not very substantial in terms of a state-of-the-art industrial solver (see the following section), though was motivated by the HWMCC timeout period; **(2)** increasing the number of solved instances by 60% is a nontrivial improvement nonetheless. Table I details the results of these additional solutions.

C. *Effect on a State-of-the-Art Verification Tool*

Due to space limits, our experiments do not detail many algorithms which are commonplace in a state-of-the-art verification tool. It is well-known that higher-quality abstractions may boost the effectiveness of a variety of these algorithms, such as the reduction capability of retiming [11]. The fact that our limited experiments demonstrate verification benefits for IC3 alone is practically encouraging. When using a larger set of algorithms and a larger runtime, the benefits of higher-quality abstractions becomes even more pronounced. We additionally note that an industrial-strength multiple-algorithm tool likely would use a strategy of running IC3 for a small time-bound early in its strategy, to rule out simpler problems. Extracting localization hints from those runs has virtually no overhead, yet may immediately yield higher-quality abstractions. We also note that we have tuned our bounded model checking-based abstraction over years of industrial application, whereas the use of IC3 hints is much less mature and we are hopeful to discover improved invariants with continued experience.

D. *Justification for Effectiveness*

A natural question is why IC3 hint-guided localization yields better abstractions than bounded model checking alone. We provide two insights: **(1)** just as CBA and PBA complement each other to yield smaller abstractions, IC3 offers yet another qualitatively-distinct heuristic guidance to complement these techniques. **(2)** There are commonalities in how bounded model checking and IC3 attempt to justify a property failure via backward analysis of the design. Some of these justification attempts identify necessary logic; some spuriously involve unnecessary logic. IC3 in a sense performs additional filtering of the impact of the spurious justification attempts, by requiring forward reachability analysis to generate those invariants only where a potential failure justification may be eliminated. In contrast, justifications that cannot be eliminated by reachability analysis will not yield invariants and thus be less likely to bloat the abstraction.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrate how the invariants generated by an incomplete IC3 run may be used to generate higher-quality localization abstractions. We furthermore demonstrate that the improved abstractions enhance verification using IC3 itself, and have noted even greater benefits from the higher-quality abstractions using heavier-weight verification flows. Areas of ongoing work include improving heuristics for prioritizing state variables given IC3 information, and to explore methods to prune irrelevant invariants that IC3 is tuned to aggressively propagate. Another direction is to additionally explore the use of IC3 hints on proof-based abstraction.

REFERENCES

[1] P. Chauhan et al., "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *FMCAD*, 2002.

[2] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *TACAS*, April 2004.

[3] N. Amla and K. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *FMCAD*, Nov. 2004.

[4] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Jan. 2011.

[5] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, Nov. 2011.

[6] N. Eén, A. Mishchenko, and N. Amla, "A single-instance incremental sat formulation of proof- and counterexample-based abstraction," in *FMCAD*, 2010.

[7] D. Wang, P.-H. Ho, J. Long, J. H. Kukula, Y. Zhu, H.-K. T. Ma, and R. F. Damiano, "Formal property verification by abstraction refinement with formal, simulation and hybrid engines," in *DAC*, June 2001.

[8] B. Li and F. Somenzi, "Efficient computation of small abstraction refinements," in *ICCD*, Nov. 2004.

[9] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[10] Hardware Model Checking Competition 2011. http://fmv.jku.at/hwmcc11.

[11] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *CHARME*, Oct. 2005.

Formal for Everyone – Challenges in Achievable Multicore Design and Verification

Daryl Stewart

ARM, Cambridge, United Kingdom

Abstract

Since the introduction of the ARM11 MP, an increasing number of ARM's products have been multicore-capable. Design and verification engineers must now cope with a myriad of interdependent behaviours and requirements – how does weak memory ordering affect system coherency; can I simulate four, sixteen or more cores; what about heterogeneity; what guarantees does lock free code require of the interconnect; how do I even describe barriers; what about the effect of power domains; can I safely introduce non-determinism as a by-product of optimisation; does my architectural specification guarantee deadlock freedom and forward progress; does my design implement my architectural specification?

We will give a brief impression of the complexity of this task, and describe progress on open problems in ensuring that future multicore systems are not prohibitively difficult or expensive for mere mortals to develop.

A quantifier-free SMT encoding of non-linear hybrid automata

Alessandro Cimatti
Fondazione Bruno Kessler
Email: cimatti@fbk.eu

Sergio Mover
Fondazione Bruno Kessler
Email: mover@fbk.eu

Stefano Tonetta
Fondazione Bruno Kessler
Email: tonettas@fbk.eu

***Abstract*—Hybrid systems are a clean modeling framework for embedded systems, which feature integrated discrete and continuous dynamics. A well-known source of complexity comes from the time invariants, which represent an implicit quantification of a constraint over all time points of a continuous transition.**

Emerging techniques based on Satisfiability Modulo Theory (SMT) have been found promising for the verification and validation of hybrid systems because they combine discrete reasoning with solvers for first-order theories. However, these techniques are efficient for quantifier-free theories and the current approaches have so far either ignored time invariants or have been limited to linear hybrid automata[1].

In this paper, we propose a new method that encodes a class of hybrid systems into transition systems with quantifier-free formulas. The method does not rely on expensive quantifier elimination procedures. Rather, it exploits the sequential nature of the transition system to split the continuous evolution enforcing the invariants on the discrete time points. This pushes the application of SMT-based techniques beyond the standard linear case.

I. Introduction

Embedded systems (e.g. control systems for railways, avionics, and space) feature the interaction of discrete systems with the environment by means of controlled and monitored variables that evolve continuously in time. The validation and verification of embedded systems designs must often take into account a model of the continuous evolution of such variables. *Hybrid systems* [3] are a clean modeling framework for embedded systems because they exhibit both continuous transitions ruled by flow conditions (i.e., constraints on the derivatives of continuous variables) and discrete changes represented with logical formulas. A fundamental step in the design of these systems is the validation and verification of the models, performed by checking properties such as invariants or reachability. In spite of the undecidability of these problems, several verification techniques have been developed and have proved to be applicable in a wide number of cases. Among these techniques, common approaches are the computation of the reachable states, and the use of abstraction or deduction systems (see [2] for a recent survey).

An emerging approach to the verification of hybrid systems is the application of verification techniques based on SMT [5]. The hybrid system is encoded into a symbolic transition system and reachability problems are represented by means of first-order formulas. The encoding allows the application of general-purpose SAT-based verification techniques such as Bounded Model Checking (BMC) [6], interpolation-based model checking [27], k-induction [32], and predicate abstraction [20]. Examples of such SMT-based approaches are [4], [1], [24], [22], [17], [25], [18], [23]. Specific techniques have also been proposed for networks of hybrid systems [9], [11], [13], and for requirements [14]. Also thanks to the strong progress in the field of SMT, these approaches are increasingly applied in real settings (e.g. the design of complex space systems [7], [8], [34]).

A well-known problem of this approach is the encoding of invariants. In order to preserve the semantics of the hybrid system, the formula representing a continuous (timed) transition between two time points t and t' must guarantee that the invariant holds along all points of the implicit continuous evolution between the state $s(t)$ and the state $s(t')$. A straightforward approach would create a quantified formula which treats the invariant as a formula $Inv(t)$ over the variable representing real time and quantifies the formula along all time points of the timed transition, i.e., $\forall\epsilon \in [t, t'], Inv(\epsilon)$. In general, it is an open question how to handle such quantifiers (see for example [1], [18]): the elimination of quantifiers is in general not possible, and when the elimination is theoretically possible (such as in the case of the theory of reals, i.e., polynomial constraints) it is in practice not feasible beyond the quadratic case. Only in particular cases (such as when the continuous evolution of variables is linear in time, and Inv is convex), the encoding is equivalent to the quantifier-free formula $Inv(t) \wedge Inv(t')$.

In this paper, we propose a new approach to efficiently encode invariants as quantifier-free formulas. Intuitively, the encoding can be thought of as generalizing the linear case, forcing the invariant before and after the timed transition ($Inv(t) \wedge Inv(t')$), and imposing the derivative of the invariant to be constant in sign throughout the timed transition. This reduces the invariant to a quantified formula over the derivatives of the continuous variables. Applying these reduction recursively, in some cases, one may obtain a quantifier-free encoding. This is guaranteed, for example, in the case of polynomial hybrid automata, where the derivatives eventually reduce to zero. We obtain a quantifier-free encoding also in interesting cases of non-linear hybrid automata with

[1]In the context of this paper, the terms "linear", "non-linear", and "polynomial" refer to the formulas over the time variable used to describe continuous and discrete transitions, and not to the type of ODE.

transcendental functions. As a result, a key contribution of the paper is a quantifier-free encoding of polynomial hybrid automata, which enables the application of SMT-based verification techniques to a broader class of hybrid systems. The approach has been implemented and evaluated on a set of benchmarks. The analysis shows that the proposed technique allows us to solve problems where an abstraction that simply ignores the invariants is too coarse to guarantee soundness and completeness.

The rest of this paper is structured as follows. In Sec. II we present some background. In Sec. III and IV we present the encoding, together with the statement of correctness. A comparison with related work is described in Section V, whilst the experimental evaluation is presented in Section VI. In Section VII we draw some conclusions, and outline directions for future work.

II. Background

A. First-order Transition Systems

Given a set V of variables, we denote with $V', \dot{V}, V^0, V^1, \dots$ copies of such set. Given a first-order signature Σ, a first-order Σ-Transition System (TS) is a tuple $S = \langle V, Init, Inv, Trans \rangle$ such that:

- V is a set of variables;
- $Init$ is a first-order Σ-formula over V (called initial condition);
- Inv is a first-order Σ-formula over V (called invariant condition);
- $Trans$ is a first-order Σ-formula over $V \cup V'$ (called transition condition).

Let $\Sigma_{\mathbb{R}}$ be the standard signature of real ordered field. In the following, we will consider signatures Σ that are extensions of $\Sigma_{\mathbb{R}}$, the structure $\overline{\mathbb{R}}$ of the real ordered field extended with transcendental functions such as the exponential and the trigonometric functions, and formulas will be interpreted in an appropriate extension of the first-order theory of the real numbers for such structure $\overline{\mathbb{R}}$.

A *state* s is an assignment to the variables V. We denote with $s', \dot{s}, s^0, s^1, \dots$ the corresponding assignment to the copy $V', \dot{V}, V^0, V^1, \dots$ of V.

A sequence $s_0, s_1, \dots, s_k$ of states is a model (also called *path*) of the transition system $S = \langle V, Init, Inv, Trans \rangle$ iff:

- s_0 satisfies $Init$;
- for every $0 \le i \le k$, s_i satisfies Inv;
- for every $0 \le i < k$, s_i, s'_{i+1} satisfy $Trans$.

Many verification techniques for transition systems such as Bounded Model Checking (BMC) [6] are based on satisfiability checking interacting with queries to SAT/SMT solvers.

B. Hybrid traces

We denote with $\dot{f}$ the derivative of a real function f. Let I be an interval of $\mathbb{R}$ or $\mathbb{N}$; we denote with $le(I)$ and $ue(I)$ the lower and upper endpoints of I, respectively. We denote with $\mathbb{R}^+$ the set of non-negative real numbers.

Hybrid traces [15], [14] describe the evolution of variables in every point of time. Such evolution is allowed to have a countable number of discontinuous points corresponding to changes in the discrete part of the model.

A *hybrid trace* over discrete variables V and continuous variables X is a sequence $\langle \overline{f}, \overline{I} \rangle := \langle f_0, I_0 \rangle, \langle f_1, I_1 \rangle, \dots, \langle f_k, I_k \rangle$ such that, for all i, $0 \le i \le k$,

- the intervals are adjacent, i.e. $ue(I_i) = le(I_{i+1})$;
- $le(I_0) = 0$ and I_k is right closed;
- $f_i : V \cup X \to \mathbb{R} \to \mathbb{R}$ is a function such that, for all $x \in X$, $f_i(x)$ is differentiable, and for all $v \in V$, $f_i(v)$ is constant;
- if I_i is left open [right open] and $le(I_i) = t$ [$ue(I_i) = t$] then, for all $v \in V \cup X$, $f_i(v)(t) = f_{i-1}(v)(t)$, [$f_i(v)(t) = f_{i+1}(v)(t)$].

We say that a trace is a sampling refinement of another one if it has been obtained by splitting an open interval into two parts by adding a sampling point in the middle [15]. A *partitioning function* μ is a sequence $\mu_0, \mu_1, \mu_2, \dots$ of non-empty, adjacent and disjoint intervals of $\mathbb{N}$ partitioning $\mathbb{N}$. Formally, $\bigcup_{i \in \mathbb{N}} \mu_i = \mathbb{N}$ and $ue(\mu_i) = le(\mu_{i+1}) - 1$. A hybrid trace $\langle \overline{f}', \overline{I}' \rangle$ is a *sampling refinement* of $\langle \overline{f}, \overline{I} \rangle$ (denoted with $\langle \overline{f}', \overline{I}' \rangle \preceq \langle \overline{f}, \overline{I} \rangle$) iff, there exists a partitioning μ such that for all $i \in \mathbb{N}$, $I_i = \bigcup_{j \in \mu_i} I'_j$ and, for all $j \in \mu_j$, $f'_j = f_i$. We extend the relation to set L_1 and L_2 of traces as follows: $L_1 \preceq L_2$ iff for every trace $\sigma_2 \in L_2$ there exists $\sigma_1 \in L_1$ such that $\sigma_1 \preceq \sigma_2$.

In the paper, we will assume that the evolution of predicates along time have the finite variability property: we say that a predicate $P(t)$ over a real variable t has *finite variability* [30] iff for any bounded interval J there exists a finite sequence of real numbers $t_0 < \dots < t_n$ such that $t_0 = le(J)$, $t_n = ue(J)$, and for all $i \in [1, n]$, either for all $\epsilon \in (t_{i-1}, t_i)$, $P(\epsilon)$ or for all $\epsilon \in (t_{i-1}, t_i)$, $\neg P(\epsilon)$. The last condition means that the predicate is constant in the interval (t_{i-1}, t_i). If P is in the form $g(t) \bowtie 0$ with g continuous and $\bowtie \in \{\ge, \le, <, >\}$, in the points in which P changes value, $g(t) = 0$. Thus, $g \bowtie 0$ has finite variability iff for any bounded interval J there exists a finite sequence of real numbers $t_0 < \dots < t_n$ such that $t_0 = le(J)$, $t_n = ue(J)$, and for all $i \in [1, n]$, either for all $\epsilon \in [t_{i-1}, t_i]$, $g(\epsilon) \ge 0$ or for all $\epsilon \in [t_{i-1}, t_i]$, $g(\epsilon) \le 0$. We denote this condition with *Constant*(P, t_{i-1}, t_i).

Proposition 1: Assuming that a predicate P has finite variability, for every hybrid trace σ, there exists a sampling refinement of σ for which which P is constant in the open part of every interval.

C. Hybrid systems

Hybrid systems [3] extend transition systems with continuous dynamics. A *Hybrid System* (HS) is a tuple $\langle V, X, Init, Trans, Inv, Flow \rangle$ where:

- V is the set of discrete variables,
- X is the set of continuous variables,
- $Init$ is a $\Sigma_{\mathbb{R}}$-formula over $V \cup X$ (called the initial condition);
- Inv is a $\Sigma_{\mathbb{R}}$-formula over $V \cup X$ (called the invariant condition).

- $Trans$ is a $\Sigma_{\mathbb{R}}$-formula over $V \cup X \cup V' \cup X'$ (called the transition condition);
- $Flow$ is a $\Sigma_{\mathbb{R}}$-formula over $V \cup X \cup \dot{X}$ (called the flow condition).

Given a hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$, we denote with $s_{f_i}(t)$ the state assigning to every variable $v \in V \cup X$ the value $f_i(v)(t)$ and with $\dot{s}_{f_i}(t)$ the assignment that maps every variable $v \in X$ with the value $\dot{f}_i(v)(t)$.

A hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ is a model (also called *path*) of the HS $S = \langle V, X, Init, Inv, Trans, Flow\rangle$ iff:

- $s_{f_0}(0)$ satisfies $Init$;
- for every $0 \leq i \leq k$, for all $t \in I_i$, $s_{f_i}(t)$ satisfies Inv;
- for every $0 \leq i < k$, if I_i is right closed with $ue(I_i) = t$ and I_{i+1} is left closed with $le(I_{i+1}) = t'$, then $s_{f_i}(t), s'_{f_{i+1}}(t')$ satisfies $Trans$;
- for every $0 \leq i \leq k$, for all $t \in I_i$, $s_{f_i}(t), \dot{s}_{f_i}(t)$ satisfy $Flow$.

The *language* $L(S)$ is the set of models of S.

Proposition 2: A sampling refinement of a path of an HS S is a path of S too.

Intuitively, sampling refinement just splits an interval into sub-intervals and therefore does not change either the initial state or the discrete transitions. Thus, the conditions remain satisfied by the corresponding points.

Sampling refinement preserves reachability properties in the sense that if $L' \preceq L(S)$ then there exists a trace in L' reaching a condition ϕ iff there exists a trace in $L(S)$ reaching ϕ (similarly for LTL properties without next operators [15] and HRELTL properties [14]).

Remark 1: In the above definition, the flow conditions are general predicates over the derivatives of X. In the following, we are considering HSs with continuous dynamics described by ODEs in form $\dot{X} = F(X)$ (i.e., for all $x \in X$, $\dot{x} = F_x(X)$).

D. Encoding of hybrid into transition systems

In this section, we show a standard encoding of HSs into a transition system with formulas over the reals. In general, for encoding, we mean a transition system that preserves the properties of interest. In this paper, we say that the transition system is an encoding of a HS if it represents its language or a sampling refinement thereof (thus preserving for example reachability).

In this encoding, we assume that the system of ODEs admits a primitive solution $f(V, t)$, which is uniquely determined by the state at the beginning of the timed transition. Moreover, we assume that the time intervals of the hybrid traces satisfying the HSs are all in the form either $[t_1, t_2)$ (i.e., left closed, right open) or $[t_1, t_1]$ (i.e., singular intervals). This simplifies the encoding but a more general encoding is possible (see for example [14]). Note also that the restriction does not affect the validity of Proposition 1, which regards only the open parts of the intervals.

Theorem 1: Given a HS S, there exists a TS S_D such that there exists a one-to-one mapping between the paths of S and the paths of S_D.

We call S_D the *encoding* of S.

Sketched proof: We encode a HS S in the TS $S_D = \langle V_D, Init_D, Inv_D, Trans_D\rangle$ where:

- $V_D := V \cup X \cup \{t\}$
  (t is a real variable that stores the current real time of the system).
- $Init_D := t = 0 \wedge Init$.
- $Inv_D := Inv$
  (note that this does not guarantee that the invariants of S hold for the entire duration of a continuous transition. This is taken into account in $Trans_D$).
- $Trans_D :=$ TIMED $\vee$ UNTIMED where
  - TIMED $:= t' > t \wedge V' = V \wedge X' = f(V \cup X, t') \wedge \forall \epsilon \in [t, t'], Inv(V, f(\epsilon))$
  - UNTIMED $:= t' = t \wedge Trans(V, X, V', X')$.

Let the hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ be a path of S. Then, the sequence of states $f_0(le(I_0)), f_1(le(I_1)), \ldots, f_k(le(I_k))$ is a path of S_D.

Let the sequence $s_0, s_1, \ldots, s_k$ be a path of S_D. Let us consider, for all $i \in [1, k]$, $f_i(v)(t) = f(s_i, t)(v)$. Let us define $I_i := [s_i(t), s_{i+1}(t))$ if $i < k$ and $s_{i+1}(t) > s_i(t)$, $I_i := [s_i(t), s_i(t)]$ if $i < k$ and $s_{i+1}(t) = s_i(t)$ or if $i = k$. Then, the hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ is a path of S. ■

Remark 2: Notice that the timed transition involves a quantified sub-formula to encode that the invariant holds along each instant of the continuous evolution. This is an issue for using standard SMT solvers which typically handle quantifier-free formulas or are not complete for quantifiers (even if the full theory with quantifiers is theoretically decidable). When the primitive solution is known and is expressed in the theory of reals (a polynomial), the quantifier can be removed to yield an equivalent quantifier-free encoding. However, in practice, this solution is not feasible beyond the quadratic case.

Remark 3: It is usually very useful to strictly alternate timed and discrete transitions to simplify the encoding and improve the search (see e.g. [1]). The encoding of Theorem 1 does not force such alternation, to enable other forms of simplification. In the following, we will clarify when we use alternation.

Hereafter, we assume that every universal quantifier occurs positively in $Trans_D$ and that it is in the form $\forall \epsilon \in [t, t'], g(\epsilon) \bowtie 0$ with $\bowtie \in \{<, \leq, >, \geq, =\}$. As shown in Appendix A, we can generalize the approach to generic formulas.

III. REMOVING QUANTIFIERS FROM THE INVARIANTS

A. Reduction to flow invariants

In this section we present the main theorem of the paper. The goal of the theorem is to reduce the quantified formula of an invariant to a quantified formula over its derivatives. In some cases, this simplifies the quantified formula.

The following theorems assume the finite variability of predicates of the derivatives. Many functions have this property, in particular polynomials and some simple transcendental functions.

Theorem 2: If $g : \mathbb{R} \to \mathbb{R}$ is a differentiable function and $\dot{g} \bowtie 0$ ($\bowtie \in \{\geq, >, \leq, <\}$) has finite variability, then $\forall \epsilon \in [t, t'], g(\epsilon) \bowtie 0$ iff there exists a finite sequence of real numbers $t = t_0 < \ldots < t_n = t'$ such that $\bigwedge_{0 \leq i \leq n} g(t_i) \bowtie 0 \wedge \bigwedge_{0 < i \leq n}$ *Constant*$(\dot{g} \geq 0, t_{i-1}, t_i)$.

Proof: Let us assume that $\bowtie \in \{\geq, >\}$.

($\Rightarrow$) Since $\dot{g} \bowtie 0$ has finite variability, there exists a finite sequence of real numbers $t_0 = t < \ldots < t_n = t'$ such that $\bigwedge_{0<i\leq n}$ *Constant*$(\dot{g} \geq 0, t_{i-1}, t_i)$ by definition. Moreover, since $\forall \epsilon \in [t, t'], g(\epsilon) \bowtie 0$, $g \bowtie 0$ holds also in the time points $t_0, \ldots, t_n$.

($\Leftarrow$) Assume by contradiction that there exists $t_b \in [t, t']$ such that $g(t_b) \bowtie 0$ is false. Since $\bigwedge_{0\leq i\leq n} g(t_i) \bowtie 0$, there exists $i \in [1, n]$ such that $t_b \in (t_{i-1}, t_i)$. Since g is differentiable, by the mean value theorem, there exists a point $t'_b \in (t_{i-1}, t_b)$ such that $\dot{g}(t'_b) = \frac{g(t_b)-g(t_{i-1})}{(t_b-t_{i-1})}$ and therefore $\dot{g}(t'_b) < 0$. Similarly, there exists a point $t''_b \in (t_b, t_i)$ such that $\dot{g}(t''_b) = \frac{g(t_i)-g(t_b)}{(t_i-t_b)}$ and therefore $\dot{g}(t''_b) > 0$. Thus, $\dot{g}$ is not constant over (t_{i-1}, t_i) contradicting the hypothesis. We conclude that $\forall \epsilon \in [t, t'], g(\epsilon) \bowtie 0$.

The cases in which $\bowtie \in \{\leq, <\}$ can be proved similarly. ■

When the predicate is an equality, the reduction is simpler.

Corollary 1: If $g : \mathbb{R} \to \mathbb{R}$ is a differentiable function and $\dot{g} = 0$ has finite variability, then $\forall \epsilon \in [t, t'], g(\epsilon) = 0$ iff $g(t) = 0 \wedge g(t') = 0 \wedge \forall \epsilon \in [t, t'], \dot{g}(\epsilon) = 0$.

The definition of *Constant*() contains quantified subformulas in the form $\forall \epsilon \in [t, t'], \dot{g} \bowtie 0$. Therefore, the reduction can be iterated trying to remove the quantifiers.

Theorem 2 can be used to simplify the encoding of the invariant of an HS. Let the invariant be in the form $g(X) \bowtie 0$ ($\bowtie \in \{\geq, \leq, >, <, =\}$). Let $f : \mathbb{R} \to \mathbb{R}^{|X|}$ be the solution of the flow condition. If f and g are differentiable functions and $\frac{d}{dt}(g \circ f) \bowtie 0$ has finite variability, then $\forall \epsilon \in [t, t'], g(f(\epsilon)) \bowtie 0$ iff there exists a finite sequence of real numbers $t_0 = t < \ldots < t_n = t'$ such that $\bigwedge_{0\leq i\leq n} g(f(t_i)) \bowtie 0 \wedge \bigwedge_{0<i\leq n}$ *Constant*$(\frac{d}{dt}(g \circ f) \geq 0, t_{i-1}, t_i)$.

The geometrical interpretation of $\frac{d}{dt}(g \circ f)$ is the scalar product of the gradient of the curve g and the derivative vector $\dot{f}$: in fact, $\frac{d}{dt} g(f(t)) = \nabla g \cdot \dot{f}$ where $\nabla g = \langle \frac{\partial g}{\partial x_1}, \ldots, \frac{\partial g}{\partial x_n} \rangle$. Therefore, in the theorem, the condition of $\dot{g} \geq 0$ of being constant in the interval means that the function f is uniformly getting closer to (or farther from) the curve g in that interval.

As a side note, in the case of ODEs $\dot{X} = F(X)$, the new quantified formula $\forall \epsilon \in [t, t'], \frac{d}{dt}(g \circ f) \geq 0$ is equivalent to the invariant $\nabla g \cdot F \geq 0$. Thus, the reduction can be also applied without need of the primitive solutions.

In the case that the invariants are polynomial and the continuous variables are polynomial functions of time, the derivative will eventually reduce to zero.

B. Applications

1) Application to polynomial hybrid automata: We consider the class of HS where the invariants and the primitive solution of the ODEs are polynomial functions of time (see also [19]). The polynomial may contain some discrete variables as coefficients to account for uncertainties in the inputs, model parameters, etc. Note that several classes of HS with linear ODE can be expressed as a polynomial hybrid automaton. This is because the primitive solution to the ODEs can be expressed as a quantifier free formula in the theory of reals for several classes of linear systems [26][2].

Theorem 3: The invariant of a polynomial hybrid automata can be encoded with a quantifier-free formula.

Proof: In the case of polynomial hybrid automata, the invariant $g \bowtie 0$ is encoded into a formula in the form $\forall \epsilon \in [t, t'], g(f(\epsilon)) \bowtie 0$. If g and f are polynomials, $g \circ f$ is also a polynomial. The derivative of a polynomial has a lower degree than the polynomial itself. Thus, at every application of Theorem 2, the degree of the polynomial inside the quantifier strictly decreases. Thus, after a finite number of applications of the theorem, we obtain a quantifier-free formula. ■

Example 1: Let us consider the classical example of the bouncing ball. Suppose the ball moves in two dimensions x and y, where x is the horizontal coordinate, with $\dot{x} = v_0$, and y is the vertical coordinate, with $\dot{y} = w$ and $\dot{w} = -g$. Thus, the primitive solution is $x(t) = v_0 t + x_0$, $y(t) = -\frac{g}{2}t^2 + w_0 t + y_0$, and $w(t) = -gt + w_0$. Suppose the ball is bouncing on a parabolic hill, a curved surface with equation $y + ax^2 + bx + c = 0$. The invariant of the continuous transition is $y + ax^2 + bx + c \geq 0$ and its encoding is $\forall \epsilon \in [t, t'], y(\epsilon) + ax^2(\epsilon) + bx(\epsilon) + c \geq 0$, which is quadratic in ϵ. After applying the Theorem 2 twice, we obtain the following quantifier-free formula: $y(t) + ax^2(t) + bx(t) + c \geq 0 \wedge$
$y(t_1) + ax^2(t_1) + bx(t_1) + c \geq 0 \wedge$
$y(t') + ax^2(t') + bx(t') + c \geq 0 \wedge$
$((w(t) + 2av_0x(t) + bv_0 \geq 0 \wedge w(t_1) + 2av_0x(t_1) + bv_0 \geq 0) \vee$
$(w(t) + 2av_0x(t) + bv_0 \leq 0 \wedge w(t_1) + 2av_0x(t_1) + bv_0 \leq 0)) \wedge$
$((w(t_1) + 2av_0x(t_1) + bv_0 \geq 0 \wedge w(t') + 2av_0x(t') + bv_0 \geq 0) \vee$
$(w(t_1) + 2av_0x(t_1) + bv_0 \leq 0 \wedge w(t') + 2av_0x(t') + bv_0 \leq 0))$

2) Application to non-linear hybrid automata: In the general case of non-linear hybrid automata (here meant as hybrid systems with non-polynomial functions), the reduction of Theorem 2 may result in more complex quantified formulas. Even if we restrict to polynomial invariants, their composition with transcendental primitive solutions may yield complex derivatives. However, in many cases, we can convert the derived quantified formula into a polynomial which is simpler than the original[3].

Example 2: Let us consider a temperature controller. The system is parameterized by the lower and upper temperature limits m and M, the outside temperature u, the rate b of temperature exchanged with the outside, the rate c of temperature increase due to the heater. The constraints on the parameters

[2]In particular, given a linear system $\dot{X} = AX + BU$, the reachability problem can be expressed in the theory of reals if the matrix A has a particular structure: A is nilpotent, A is diagonalizable with all rational eigenvalues, A is diagonalizable with all imaginary eigenvalues. While in the first case obtaining a primitive solution in the theory of reals is straightforward also in the presence of symbolic coefficients of the matrix, the other two cases are more involved and require to perform several substitutions to remove exponential or trigonometric functions, which assume to have constant coefficient. We refer the reader to [26] for the details.

[3]This conversion is not currently automated.

are $u < m < M \wedge c > 0 \wedge b > 0$. The HS is defined as follows:

- $V = \{h\}$ where h is a variable representing the heater.
- $X = \{x\}$ where x represents the temperature.
- $Init := m \leq x \leq M$.
- $Inv := (h = 0 \rightarrow x \geq m) \wedge (h = c \rightarrow x \leq M)$.
- $Trans := (h = 0 \rightarrow (x = m \wedge h' = c)) \wedge (h = c \rightarrow (x = M \wedge h' = 0)) \wedge x' = x$.
- $Flow := \dot{x} = b(u - x) + h$.

The primitive of the ODE is $x(t) := u + \frac{(x(0)-u)}{b}e^{(-b*t)} + \frac{c}{b}$. Its derivative is $x(t) := -(x(0) - u)e^{(-b*t)}$, which never changes sign. Therefore, applying Theorem 2, $\forall\epsilon \in [t, t'], x \geq m$ is translated into the formula $x(t) \geq m \wedge x(t') \geq m$ and similarly for $\forall\epsilon \in [t, t'], x \leq M$.

Example 3: Consider the Traffic Collision Avoidance System (TCAS) example (cfr. e.g. [29]). The continuous dynamics of a safe circular maneuver is described by the following equations $\dot{x_1} = d_1, \dot{x_2} = d_2, \dot{d_1} = -\omega d_2, \dot{d_2} = \omega d_1, \dot{y_1} = e_1, \dot{y_2} = e_2, \dot{e_1} = -\rho e_2, \dot{e_2} = \rho e_1, (x_1 - y_1)^2 + (x_2 - y_2)^2 \geq p^2$.

The primitive solution of the differential equations are:

$$\begin{aligned} x_1 = \frac{1}{\omega}sin(\theta),\ x_2 = -\frac{1}{\omega}cos(\theta), \\ d_1 = cos(\theta),\ d_2 = sin(\theta),\ \theta = \omega t + t_0, \\ y_1 = \frac{1}{\rho}sin(\xi),\ y_2 = -\frac{1}{\rho}cos(\xi), \\ e_1 = cos(\xi),\ e_2 = sin(\xi),\ \xi = \rho t + t_0 \end{aligned}$$

Substituting the primitive solution into the invariant $(x_1 - y_1)^2 + (x_2 - y_2)^2 \geq p^2$ we obtain the formula:

$$\frac{1}{\omega^2} + \frac{1}{\rho^2} - \frac{2}{\omega\rho}sin(\theta)sin(\xi) - \frac{2}{\omega\rho}cos(\theta)cos(\xi) \geq p^2.$$

which can be rewritten into: $\phi := \frac{1}{\omega^2} + \frac{1}{\rho^2} - \frac{2}{\omega\rho}cos(\theta - \xi) \geq p^2$.

The standard quantified encoding is $\forall t \in [0, \delta], \phi(t)$. Applying Theorem 2, we obtain the formula:

$$\begin{aligned} \phi(0) \wedge \phi(\delta) \wedge \quad & (\forall t(-sin(\theta - \xi)(\omega - \rho) \geq 0) \vee \\ & \forall t(-sin(\theta - \xi)(\omega - \rho) \leq 0)). \end{aligned}$$

The quantified sub-formulas can be rewritten into polynomials over θ and ξ. For example, $\forall t(-sin(\theta - \xi)(\omega - \rho) \geq 0)$ can be rewritten into $\forall t(\omega - \xi \geq 0 \wedge (\pi \leq \theta - \rho \leq 2\pi) \vee \omega - \xi \leq 0 \wedge (0 \leq \theta - \rho \leq \pi))$. Since θ and ρ are linear, this can be converted into an equivalent quantifier-free one.

IV. Encoding Polynomial HS into Transition Systems

In this section, we show how Theorem 2 can be exploited to automatically encode a polynomial HS into a transition system with quantifier-free formulas.

A. Sequential encoding

Theorem 2 states the existence of the points $t_1, \ldots, t_n$ where the derivative changes sign. However, such points are unknown. The encoding of a HS into a transition system must thus implicitly represent when the derivative of the invariant changes sign. This is achieved by simply forcing that the sign of the derivative is constant throughout the timed transition. The encoding implicitly concatenates timed transitions one after the other, delegating to the search the task of finding the sequence of time points that split the interval, so that the sign of the derivative is uniformly constant in the resulting trace.

Given a formula T including the invariant condition $\forall\epsilon \in [t, t'], g(\epsilon) \bowtie 0$, the condition can be locally replaced with $g(t) \bowtie 0 \wedge g(t') \bowtie 0 \wedge \mathit{Constant}(\dot{g}, t, t')$ obtaining a new formula $\tau(T)$.

τ performs a recursive substitution of the quantified expressions. The recursion terminates when the quantified formula is a linear polynomial, thus allowing to trivially remove the quantifiers. τ is defined recursively as follows:

$$\begin{aligned} \tau(\psi_1 \wedge \psi_2) &:= \tau(\psi_1) \wedge \tau(\psi_2) \\ \tau(\psi_1 \vee \psi_2) &:= \tau(\psi_1) \vee \tau(\psi_2) \\ \tau(\neg\psi) &:= \neg\psi, (\psi \text{ is a predicate}) \\ \tau(\forall\epsilon \in [t, t'], g(\epsilon) \bowtie 0) &:= \begin{cases} g(t) \bowtie 0 \wedge g(t') \bowtie 0 & \text{if } g \text{ linear} \\ g(t) \bowtie 0 \wedge g(t') \bowtie 0 \wedge \\ \tau(\mathit{Constant}(\dot{g}, t, t')) & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

The correctness of the transformation is given by the following theorem.

Theorem 4: If S_D is the encoding of the HS S and $\tau(S_D)$ is the transition system obtained by replacing $Trans$ with $\tau(Trans)$, then $\tau(S_D)$ is the encoding of a sampling refinement of S.

Proof: ($\Leftarrow$) If a sequence of states satisfies $\tau(S_D)$, then by Theorem 2, the sequence satisfies also S_D, and by Theorem 1, it represents a path of S. ($\Rightarrow$) Consider a hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ which is a path of S. Assuming that $\dot{g}$ has finite variability, we can refine the hybrid trace into a new hybrid trace in which $\dot{g}$ is constant in every interval. The new hybrid trace also satisfies S by Theorem 2 and thus the corresponding discrete trace $s_0, \ldots, s_k$ satisfies its encoding S_D. At every i, if s_i satisfies $\forall\epsilon \in [t, t'], g(\epsilon) \bowtie 0$, then both $f(s_i, t)$ and $f(s_i, t')$ satisfy $g \bowtie 0$. Since $\dot{g}$ has constant sign in I_i, s_i satisfies also $\tau(Trans)$. Therefore the discrete trace satisfies also $\tau(S_D)$. ■

The recursive definition of τ in (1) creates a formula whose size is exponential in the degree of the polynomial inside the invariant. We use the following equivalence to keep the size of the encoding *linear* in the degree of the polynomial (here g is not linear):

$$\begin{aligned} \tau(\mathit{Constant}(g, t, t')) &= (g(t) \geq 0 \wedge g(t') \geq 0 \wedge \\ &\quad \tau(\mathit{Constant}(\dot{g}, t, t'))) \vee \\ &\quad (g(t) \leq 0 \wedge g(t') \leq 0 \wedge \\ &\quad \tau(\mathit{Constant}(\dot{g}, t, t'))) \\ &= ((g(t) \geq 0 \wedge g(t') \geq 0) \vee \\ &\quad (g(t) \leq 0 \wedge g(t') \leq 0)) \wedge \\ &\quad \tau(\mathit{Constant}(\dot{g}, t, t')) \end{aligned} \tag{2}$$

Another optimization that we implemented is the use of some *lemmas* that relate the value of polynomials to the value of their derivatives. More specifically, we optionally add to τ the following formulas:
$(\dot{g}(t) > 0 \vee \dot{g}(t') > 0) \rightarrow$
$((g(t) \geq 0 \rightarrow g(t') \geq 0) \wedge (g(t') \leq 0 \rightarrow g(t) \leq 0)) \wedge$
$(\dot{g}(t) < 0 \vee \dot{g}(t') < 0) \rightarrow$
$((g(t) \leq 0 \rightarrow g(t') \leq 0) \wedge (g(t') \geq 0 \rightarrow g(t) \geq 0))$
The formula means that when the derivative is positive g can only increase (thus cannot pass from positive to negative) and vice versa when $\dot{g}$ is negative g can only decrease (thus cannot pass from negative to positive).

B. Bound on required splitting

The sequential encoding may force the split of a continuous transition in several transitions, since the predicates introduced to remove the quantifiers forces the derivatives of the invariant conditions to be constant. While the encoding enables to remove the quantifier, the depth of the bounded model checking formula may increase due to the splitting. In incremental bounded model checking, the burden of finding how many splits are necessary is delegated to the search.

In the case of polynomial hybrid automata we can compute an upper bound on the number of consecutive continuous transitions (continuous transitions not separated by a discrete transition) needed to simulate the longest quantified continuous transition (the continuous transition with the maximum time elapse).

We can compute the upper bound on the number of intervals needed to "cover" the quantified continuous transition for the invariant predicate $\forall \epsilon \in [t, t'] g(\epsilon) \bowtie 0$. If $\Omega(g)$ is the degree of the polynomial, then the maximum number of intervals that have to be considered is $ub(g) = \frac{\Omega(g)*(\Omega(g)-1)}{2}$. In fact, the i-th derivative of g has degree $\Omega(g) - i$ and thus changes sign $\Omega(g) - i$ times.

C. Layering

In the BMC settings we usually perform a search where we check if the target is violated for an increasing path length. In principle, the removal of the quantifiers requires more continuous transitions, thus increasing the size of the formula passed to solver. It is convenient to use a "layered" approach, where we first reach the target in an over-approximation of the HS, where invariants are not guaranteed to hold, and then we check if there exists a path that reaches the target and where invariants hold.

V. Related work

The quantifier-free encoding that we propose is related to quantifier elimination procedures (see, e.g., [16]). It is not a quantifier elimination procedure in that it contains new variables that are implicitly existentially quantified. In fact, we apply the reduction even in some cases of transcendental functions. The burden to remove the quantifiers is delegated to the verification techniques if necessary. We claim that quantifier elimination is somehow an overkill: the verification techniques does not often need the precise region of points where the invariant holds; it is usually sufficient either to pick some "good" values (in case of reachability) or to find "good" invariants (in case of safety verification).

Several works focus on the reachability problem for hybrid systems, but they use less expressive invariants or they restrict the class of the analyzed hybrid automata. We extend the bounded model checking encoding of linear hybrid automata [4], [1], where invariants hold iff they hold at the first and the last instant of a timed transition, thus the resulting encoding is quantifier free. Other approaches [10], [18], [23] focus on non-linear hybrid automata. In [10], the authors solve the reachability problem for non-linear convex hybrid automata. The restriction to convex invariant and linear flow conditions, or to monotonic invariant and convex flow, allows to easily encode the invariants without quantifiers. Many examples, including those mentioned in this paper, do not fall in this class of automata. In [18] the authors propose an SMT solver modulo ODEs, that can be used to perform bounded model checking on hybrid automata where the flow conditions are ODEs. The only allowed invariants are of the form $x \in [l, u]$, where x is a continuous variable and $l, u \in \mathbb{R}$. Their main focus is on the integration of numerical methods to compute the initial value problem for ODEs, while they cannot manage more complex invariants (e.g. linear functions). ODEs are also handled directly in [23]. This is done by computing the precise intersection of the continuous flow with the guards of the hybrid automaton. The solver can in principle handle invariants, but the authors state that the implementation is not mature enough to evaluate the approach. Approaches based on motion planning [28] do not encode symbolically the invariants, since they simulate the ODEs using numerical methods. In contrast, we encode a set of continuous transitions.

The prominent approaches to the verification of HSs are based either on the exploration of the reachable states or on deductive systems. We refer the readers to [2] for a recent survey. The focus of our work is on the SMT-based paradigm, which, although less mature, seems promising.

Our settings also differs from the works that build abstractions for HSs. The approaches described in [33], [31] use techniques based on the sign of derivatives such as ours. However, the purpose is different in that they generate over-approximations of the HS.

Finally, we mention the "clock translation" described in [21], where invariants are translated into constraints on time. However, the translation is restricted to monotonic flows (plus other restrictions on the independence of variables).

VI. Experimental evaluation

We applied our approach to several benchmarks of non-linear hybrid automata, obtaining a quantifier-free encoding. For the polynomial subcase we used the ETCS benchmark [22], an industrial case study of the braking control system of trains, the classic bouncing ball, and a simple ballistics example. For the bouncing ball, we used four variants: a ball moving vertically in one dimension and bouncing on a

	# vars	Max degree	REDLOG	QEPCAD
etcs_braking	4	2	0.14	0.05
ball_1d_plain	4	2	0.10	0.03
ball_2d_plain	4	2	0.10	0.03
ball_2d_hill	5	2	0.15	T.O. $>$ 3600.00
ball_2d_slope	5	4	N.A.	T.O. $>$ 3600.00
simple_ballistics	5	4	N.A.	T.O. $>$ 3600.00

TABLE I
RESULTS OF APPLYING QUANTIFIER ELIMINATION TO THE POLYNOMIAL BENCHMARKS (MAX DEGREE IS THE MAXIMUM DEGREE OF THE QUANTIFIED VARIABLE,T.O.IS A TIME OUT OF 3600 SECONDS, N.A. MEANS NOT APPLICABLE).

	quantifier-free encoding	qelim (qepcad)	qelim (redlog)
etcs_braking	66.75 / 17	161.52 / 17	168.16 / 17
ball_1d_plain.01	0.05 / 2	0.05 / 2	0.03 / 2
ball_1d_plain.02	25.50 / 6	0.09 / 4	0.06 / 4
ball_1d_plain.03	31.43 / 10	0.28 / 6	0.40 / 6
ball_1d_plain.04	36.23 / 14	0.46 / 8	0.65 / 8
ball_1d_plain.05	151.41 / 18	1.27 / 10	1.51 / 10
ball_2d_plain.01	0.08 / 2	0.18 / 2	0.28 / 2
ball_2d_plain.02	4.20 / 6	3.14 / 6	3.64 / 6
ball_2d_plain.03	16.04 / 10	15.90 / 10	62.64 / 10
ball_2d_hill.01	1.30 / 4	na / na	0.94 / 2
ball_2d_hill.02	118.67 / 8	na / na	15.36 / 4
ball_2d_slope.01	to / na	na / na	na / na
simple_ballistics	8.31 / 1	na / na	na / na

TABLE II
RESULTS (RUNNING TIME / PATH LENGTH) OF BMC WITH THE DIFFERENT ENCODINGS.

plain floor, a two-dimensional variant with constant horizontal speed, a third variant still in two-dimensions but bouncing on a hill (vertical parabola), and a fourth variant bouncing on a slope (horizontal parabola). As for the ballistics example, we modeled an object that flies above an obstacle keeping below a certain ceiling. As for nonlinear benchmarks with transcendental functions, we used the temperature controller, the TCAS benchmark and the steering car mentioned in Sec. III-B2. All the benchmarks are publicly available at http://es.fbk.eu/people/mover/tests/FMCAD12/.

The techniques discussed in the previous sections have been implemented in an extension of NUSMV[4], which is able to deal with HSs formalized in the HYDI language [12]. The NUSMV extension features an SMT-based approach to the verification of HSs, including bounded model checking and inductive reasoning. We automatically encode the invariants for polynomial hybrid automata, while we manually encode the invariants for the other benchmarks. iSAT[5] is used as the backend to solve the resulting satisfiability queries.

We evaluated the alternative use of quantifier elimination procedures, within their range of applicability, i.e. polynomial hybrid automata. We experimented with Cylindrical Algebraic Decomposition (CAD) (using QEPCAD[6]) and Virtual Substitution (using REDLOG[7]). Table I reports, for each polynomial benchmark, the time needed to obtain a quantifier free formula of the invariants using QEPCAD and REDLOG. The Virtual Substitution approach of REDLOG can only handle formulas quantified over a quadratic variable. QEPCAD is slightly more general, but de facto less useful: the results highlight the dramatic computational complexity of the procedure (e.g. ball_2d_hill, with 5 variables, times out in one hour). Thus, the quantifier elimination approach cannot even handle the polynomial benchmarks ballistic and ball_2d_slope (in addition to the benchmarks with transcendental functions).

We used the bounded model checking functionalities enabled by our approach to validate the various models and to evaluate the performance of the invariant encoding. For each model we generated different reachability properties which are falsified by traces with an increasing length. We evaluated the encoding of the invariant by comparing the time needed to find these traces with BMC. When quantifier elimination was able to produce a result, we also compared it with our approach using the same SMT-based technique, in order to evaluate the overhead caused by the splitting. The results are shown in Table II. The encoding time of our approach is instantaneous in all cases. In the cases where quantifier elimination is feasible, the resulting encoding may induce traces with a smaller number of steps, because timed transitions must not be split. This happens for the *ball_1d_plain* and the *ball_2d_hill* benchmarks. The reduced number of steps also reduces the time needed to generate the trace.

Our approach was also able to prove a simple invariant on the ballistics example, that was beyond the applicability of SMT-based techniques. We chose as obstacle a circle shape with center in $(c, 0)$ and radius r. If the ceiling level is less than r, the object cannot clearly pass. This has been proved with NUSMV and iSAT. Ignoring the invariant along the timed transitions (keeping it only on the discrete points) allows for spurious traces that forbid the inductive proof. Note that this small example is beyond the applicability of quantifier elimination (see Table I).

Some remarks are in order. Our approach strongly depends on the availability of SMT solvers for quantifier-free theories of nonlinear arithmetic, to solve the formulas resulting from our SMT-based verification engines. To this end, we tried to use all the available solvers for nonlinear arithmetic: Z3[8], SMT-RAT[9], CVC3[10], miniSMT[11], RAHD[12], hydlogic[13], dReal[14]. and iSAT[15]. Z3 and SMT-RAT implement two complete decision procedures for the non-linear arithmetic over reals. Both solvers still do not integrate a layering with the linear arithmetic solver: in this case all the linear arithmetic constraints are handled using the non-linear solver,

[4]http://nusmv.fbk.eu/
[5]http://isat.gforge.avacs.org/
[6]http://www.usna.edu/cs/ qepcad/B/QEPCAD.html
[7]http://redlog.dolzmann.de/

[8]http://research.microsoft.com/en-us/um/redmond/projects/z3/
[9]http://smtrat.sourceforge.net/
[10]http://cs.nyu.edu/acsys/cvc3/
[11]http://cl-informatik.uibk.ac.at/software/minismt/
[12]http://homepages.inf.ed.ac.uk/s0793114/rahd/
[13]http://code.google.com/p/hydlogic/
[14]http://www.cs.cmu.edu/ sicung/dReal/
[15]http://isat.gforge.avacs.org/

thus resulting in an inefficient approach. This is the case for our BMC case studies, which have a significant part of linear constraints. Instead, CVC3 and miniSMT implement an incomplete decision procedure for non-linear arithmetic (and miniSMT is tailored only to check satisfiable formulas). As a result, these solvers turned out to return "unknown" on most of the queries generated from our benchmarks. The hydlogic system turned out to be immature, while RAHD exports functionalities that are closer to a theory solver than a full SMT solver, requiring an explicit treatment of disjunctions. iSAT and dReal differ from the other solvers, since they can also provide non-precise solutions. dReal returns an unsatisfiable answer or a satisfiable answer if the formula is satisfiable under a bounded numerical perturbations. iSAT may return "unknown" exposing the results of interval constraints propagation: it produces the intervals found in the search, if these are below a user-defined threshold, as a candidate solution. In many practical cases, this is not spurious, and represents a satisfying assignment of the formula.

Overall, despite some recent progress, our experience has shown that the field still requires additional research to deliver what our approach requires, both in terms of completeness, and performance. However, we argue that our method is valuable regardless of the current status of SMT for nonlinear arithmetic. First, we proposed a solution to a problem that was a show-stopper for SMT-based verification. In fact, we are now able to solve some benchmarks that cannot be solved by overapproximation, just forgetting about the quantified invariants. Second, we are hopeful that the field of SMT can deliver quick progress in quantifier-free nonlinear arithmetic. In fact, the development of SMT solving for non-linear arithmetic has been influenced by benchmarks from other domains (e.g. most of the SMT-LIB benchmarks in NRA are from the software domain). To this extent, we generated and submitted to the SMT-LIB a vast number of benchmarks, that will trigger additional research in practically relevant directions.

VII. Conclusions

In this paper, we tackled the problem of dealing with invariant constraints in non-linear hybrid automata in the setting of SMT-based verification. This is largely an open problem, due to the presence of the universal quantifiers required to encode that the invariant must hold throughout all time instants in delay transitions.

We proposed new methods that allow for the reduction to quantifier-free theories, at the cost of introducing additional variables. Our approach is comprehensive (deals with disjunctive invariants), encompasses a large class of hybrid systems (nonlinear polynomials), and is open to new patterns of reduction, when an algorithmic solution is not possible in general. As a result, we extend the applicability of SMT-based verification methods, and were able to verify some novel benchmark problems.

In the future, we plan to proceed along the following directions. We will experiment with the application of the proposed methods as a way to concretize the abstract paths. Then, we will generalize the approach to the analysis of networks of hybrid automata; in particular, we will exploit the locality of the splits of the continuous transitions in the local time semantics framework. We will also apply a layered approach to the analysis of non-linear constraints, where less expensive (e.g. linear) solvers are applied whenever possible before resorting to expensive but more precise nonlinear solvers such as RAHD. Finally, we will apply the proposed techniques to the analysis of requirements expressed in HRELTL logic [14]. In fact, HRELTL requires the predicates to be constant in arbitrary intervals of time.

References

[1] E. Ábrahám, B. Becker, F. Klaedtke, and M. Steffen. Optimizing Bounded Model Checking for Linear Hybrid Systems. In *VMCAI*, pages 396–412, 2005.

[2] R. Alur. Formal verification of hybrid systems. In *EMSOFT*, pages 273–278, 2011.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[4] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. *ENTCS*, 119(2):17–32, 2005.

[5] C.W. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207, 1999.

[7] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *Comput. J.*, 54(5):754–775, 2011.

[8] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri, and R. Wimmer. A Model Checker for AADL. In *CAV*, pages 562–565, 2010.

[9] L. Bu, A. Cimatti, X. Li, S. Mover, and S. Tonetta. Model Checking of Hybrid Systems using Shallow Synchronization. In *FORTE*, 2010.

[10] L. Bu, J. Zhao, and X. Li. Path-Oriented Reachability Verification of a Class of Nonlinear Hybrid Automata Using Convex Programming. In *VMCAI*, pages 78–94, 2010.

[11] A. Cimatti, S. Mover, and S. Tonetta. Efficient Scenario Verification for Hybrid Automata. In *CAV*, pages 317–332, 2011.

[12] A. Cimatti, S. Mover, and S. Tonetta. HyDI: a language for symbolic hybrid systems with discrete interaction. In *EUROMICRO-SEAA*, 2011.

[13] A. Cimatti, S. Mover, and S. Tonetta. Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems. In *FMCAD*, 2011.

[14] A. Cimatti, M. Roveri, and S. Tonetta. Requirements Validation for Hybrid Systems. In *CAV*, pages 188–203, 2009.

[15] L. de Alfaro and Z. Manna. Verification in Continuous Time by Discrete Reasoning. In *AMAST*, pages 292–306, 1995.

[16] A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice. In *Alg. Algebra and Number Theory*, pages 221–247. Springer, 1998.

[17] A. Eggers, M. Fränzle, and C. Herde. SAT Modulo ODE: A Direct SAT Approach to Hybrid Systems. In *ATVA*, pages 171–185, 2008.

[18] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle. Improving SAT Modulo ODE for Hybrid Systems Analysis by Combining Different Enclosure Methods. In *SEFM*, pages 172–187, 2011.

[19] M. Fränzle. What Will Be Eventually True of Polynomial Hybrid Automata? In *TACS*, pages 340–359, 2001.

[20] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.

[21] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic Analysis of Nonlinear Hybrid Systems. 1998.

[22] C. Herde, A. Eggers, M. Fränzle, and T. Teige. Analysis of Hybrid Systems Using HySAT. In *ICONS*, pages 196–201, 2008.

[23] 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.

[24] S. Jha, B. A. Brady, and S. A. Seshia. Symbolic Reachability Analysis of Lazy Linear Hybrid Automata. In *FORMATS*, pages 241–256, 2007.

[25] T. King and C. Barrett. Exploring and Categorizing Error Spaces using BMC and SMT. In *SMT*, 2011.

[26] Gerardo Lafferriere, George J. Pappas, and Sergio Yovine. Symbolic reachability computation for families of linear vector fields. *J. Symb. Comput.*, 32(3):231–253, 2001.

[27] K.L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.

[28] E. Plaku, L.E. Kavraki, and M.Y. Vardi. Hybrid systems: from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*, 34(2):157–182, 2009.

[29] A. Platzer and E.M. Clarke. Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study. In *FM*, pages 547–562, 2009.

[30] A.M. Rabinovich. On the Decidability of Continuous Time Specification Formalisms. *J. Log. Comput.*, 8(5):669–678, 1998.

[31] S. Sankaranarayanan and A. Tiwari. Relational abstractions for continuous and hybrid systems. In *CAV*, pages 686–702, 2011.

[32] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD*, pages 108–125, 2000.

[33] A. Tiwari. Abstractions for hybrid systems. *Formal Methods in System Design*, 32(1):57–83, 2008.

[34] Y. Yushtein, M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, Th. Noll, X. Olive, and M. Roveri. System-software co-engineering: Dependability and safety perspective. In *SMC-IT*, pages 18–25. IEEE CS Press, 2011.

APPENDIX

In this section, we explain how the method can be extended to handle generic predicates for the invariants (i.e. disjunctive invariants and open intervals). We describe the extension only in the appendix because, first, it complicates the presentation, second, in practice we do not have disjunctive invariants in the benchmarks.

A. Encoding of hybrid into transition systems

We modify the encoding of a HS into a transition system with atomic quantified formulas. First, we modify the encoding considering quantification over open intervals instead of closed intervals. Later, we will prove that we can push the quantification inside the disjunctions under the assumption of finite variability.

We redefine $S_D = \langle V_D, Init_D, Inv_D, Trans_D\rangle$ as follows:

- $V_D := V \cup X \cup \{t\}$
  (t is a real variable that stores the current real time of the system).
- $Init_D := t = 0 \wedge Init$.
- $Inv_D := Inv$.
- $Trans_D :=$ TIMED $\vee$ UNTIMED
  where
  - TIMED $:= t' > t \wedge V' = V \wedge X' = f(V \cup X, t') \wedge \forall \epsilon \in (t, t'), Inv(V, f(\epsilon))$
  - UNTIMED $:= t' = t \wedge Trans(V, X, V', X')$.

Theorem 5: There exists a one-to-one mapping between the paths of S and the paths of S_D.

We call S_D the *encoding* of S.

Sketched proof:

Let the hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ be a path of S. Then, the sequence of states $f_0(le(I_0)), f_1(le(I_1), \ldots, f_k(le(I_k)$ is a path of S_D.

Let the sequence $s_0, s_1, \ldots, s_k$ be a path of S_D. Let us consider, for all $i \in [1, k]$, $f_i(v)(t) = f(s_i, t)(v)$. Let us define $I_i := [s_i(t), s_{i+1}(t))$ if $i < k$ and $s_{i+1}(t) > s_i(t)$, $I_i := [s_i(t), s_i(t)]$ if $i < k$ and $s_{i+1} = s_i(t)$ or if $i = k$. Then, the hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ is a path of S. ■

Without loss of generality we can assume that the quantified formula in $Trans$ is atomic. This is not correct in general and exploit the particular position of the quantified sub-formula in the transition condition.

Theorem 6: Assuming that the predicates ϕ and ψ have finite variability, if we replace a formula $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \psi(\epsilon)$ with $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \forall \epsilon \in (t, t'), \psi(\epsilon)$ inside $Trans_D$, we obtain the encoding of a sampling refinement to the original HS.

Proof: Clearly, $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \forall \epsilon \in (t, t'), \psi(\epsilon)$ implies $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \psi(\epsilon)$. The opposite does not hold in general. However, consider a hybrid trace $\langle f_0, I_0\rangle, \langle f_1, I_1\rangle, \ldots, \langle f_k, I_k\rangle$ which is a path of a HS S. Assuming that the predicates ϕ and ψ have finite variability, we can refine the hybrid trace into a new hybrid trace in which ϕ and ψ are constant in every interval. The new hybrid trace also satisfies S by Proposition 2 and thus the corresponding discrete trace $s_0, \ldots, s_k$ satisfies its encoding S_D. At every i, if s_i satisfies $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \psi(\epsilon)$, then $f(s_i, \epsilon)$ satisfies $\phi \vee \psi$ for all $\epsilon \in (t, t') = (le(I_i), ue(I_i))$, and thus either ϕ or ψ (since ϕ and ψ are constant in the open part of I_i). Therefore the discrete trace satisfies also the encoding with $\forall \epsilon \in (t, t'), \phi(\epsilon) \vee \forall \epsilon \in (t, t'), \psi(\epsilon)$. ■

B. Reduction to flow invariants

Theorem 7: If $g : \mathbb{R} \to \mathbb{R}$ is a differentiable function and $\dot{g} \bowtie 0$ ($\bowtie \in \{\geq, >\}$) has finite variability, then $\forall \epsilon \in (t, t'), g \bowtie 0$ iff there exists a finite set of real numbers $t = t_0 < \ldots < t_n = t'$ such that $g(t) \geq 0 \wedge g(t') \geq 0 \wedge \bigwedge_{0<i<n} g(t_i) \bowtie 0 \wedge \bigwedge_{0<i\leq n} \mathit{Constant}(\dot{g} \geq 0, t_{i-1}, t_i)$ if $\bowtie = \geq$, $g(t) \geq 0 \wedge g(t') \geq 0 \wedge \bigwedge_{0<i<n} g(t_i) \bowtie 0 \wedge \bigwedge_{0<i\leq n} \mathit{Constant}(\dot{g} \geq 0, t_{i-1}, t_i) \wedge (g(t) = 0 \to \dot{g}(t) > 0) \wedge (g(t') = 0 \to \dot{g}(t) < 0)$, if $\bowtie = >$.

Proof: ($\Rightarrow$) Since $\dot{g} \bowtie 0$ has finite variability, there exists a finite set of real numbers $t = t_0 < \ldots < t_n = t'$ such that $\bigwedge_{0<i\leq n} \mathit{Constant}(\dot{g} \geq 0, t_{i-1}, t_i)$ by definition. Moreover, since $\forall \epsilon \in (t, t'), g \bowtie 0$, $g \bowtie 0$ holds also in the time points $t_1, \ldots, t_{n-1}$. $g(t) \geq 0$ and $g(t') \geq 0$ for the continuity of g. Finally, $(g(t) = 0 \to \dot{g}(t) > 0) \wedge (g(t') = 0 \to \dot{g}(t) < 0)$, if $\bowtie = >$.

($\Leftarrow$) Assume by contradiction that there exists $t_b \in (t, t')$ such that $g(t_b) \bowtie 0$ is false. Since $\bigwedge_{0<i<n} g(t_i) \bowtie 0$, there exists $i \in [1, n]$ such that $t_b \in (t_{i-1}, t_i)$. Let us consider first the case in which $g(t) \bowtie 0$ and $g(t') \bowtie 0$ or $i \in [2, n-1]$. Since g is differentiable, for the mean value theorem, there exists a point $t'_b \in (t_{i-1}, t_b)$ such that $\dot{g}(t'_b) = \frac{g(t_b) - g(t_{i-1})}{(t_b - t_{i-1})}$ and therefore $\dot{g}(t'_b) \bowtie 0$ is false. Similarly, there exists a point $t''_b \in (t_b, t_i)$ such that $\dot{g}(t''_b) = \frac{g(t_i) - g(t_b)}{(t_i - t_b)}$ and therefore $\dot{g}(t''_b) \bowtie 0$ is true. Thus, $\dot{g}$ is not constant over (t_{i-1}, t_i) contradicting the hypothesis. Let us now consider the case in which $\bowtie = >$, $i = 1$ and $g(t) = 0$ or $i = n$ and $g(t') = 0$. Similarly as before there exists a point that contradicts $(g(t) = 0 \to \dot{g}(t) > 0) \wedge (g(t') = 0 \to \dot{g}(t) < 0)$.

We conclude that $\forall \epsilon \in (t, t'), g(\epsilon) \bowtie 0$. ■

Piecewise Linear Modeling of Nonlinear devices for Formal Verification of Analog Circuits

Yan Zhang, Sriram Sankaranarayanan and Fabio Somenzi.
University of Colorado, Boulder, CO. Email: {yan.zhang,srirams,fabio}@colorado.edu

***Abstract*—We consider different piecewise linear (PWL) models for nonlinear devices in the context of formal DC operating point and transient analyses of analog circuits. PWL models allow us to encode a verification problem as constraints in linear arithmetic, which can be solved efficiently using modern SMT solvers. Numerous approaches to piecewise linearization are possible, including piecewise constant, simplicial piecewise linearization and canonical piecewise linearization. We address the question of which PWL modeling approach is the most suitable for formal verification by experimentally evaluating the performance of various PWL models in terms of running time and accuracy for the DC operating point and transient analyses of several analog circuits. Our results are quite surprising: piecewise constant (PWC) models, the simplest approach, seem to be the most suitable in terms of the trade-off between modeling precision and the overall analysis time. Contrary to expectations, more sophisticated device models do not necessarily provide significant gains in accuracy, and may result in increased running time. We also present evidence suggesting that PWL models may not be suitable for certain transient analyses.**

I. Introduction

In this paper, we evaluate piecewise linear models (PWL) for the verification of nonlinear analog circuits. Analog circuits are indispensable in modern integrated circuits. Although, in a typical IC design, the analog circuitry occupies a small fraction of the entire die area, its design and verification requires considerable effort compared to its digital counterpart [1]. Because analog design is error-prone, many simulations, each of which may take several hours or even several days, are needed to convince the designers that the specification is met. Even with this effort, many designs still fail to work after fabrication. Therefore, formal verification techniques for analog circuits have recently emerged, as shown in Table I. Most efforts in formal verification have focused on two problems:

- *DC Operating Point Analysis:* Satisfiability solvers are used to characterize all operating points of a circuit. In many cases, SPICE simulations may miss metastable operating points that can be captured by a formal approach [2], [3].
- *Dynamic (Transient) Analysis:* Various models of dynamical systems, including ODEs [2], [4]–[6], hybrid automata [7], hybrid Petri nets [8] and frequency domain transfer functions [9] are used to study the evolution of a circuit in time.

This work was funded in part by the US National Science Foundation (NSF) under grants CNS-1016994 and CAREER grant CNS-0953941. All opinions expressed here are those of the authors and not necessarily of the NSF.

These problems are complex due to the presence of nonlinear devices such as diodes, transistors, non-Ohmic resistors and nonlinear capacitors. While solvers for reasoning about nonlinear systems are becoming sophisticated [10]–[13], their capabilities are far exceeded by the linear arithmetic SMT solvers such as MathSAT, Yices and Z3 [14]. As a result, the problem of approximating nonlinear devices by a piecewise linear model naturally presents itself. Fortunately, PWL modeling of analog devices has been well studied by the analog circuit simulation community [15]–[17]. In this regard, a wide variety of PWL modeling approaches for transistors have been considered, including simplicial piecewise linearization [16], canonical piecewise linearization [15] and their many refinements. Recent work by Tiwary et al. [2] uses simple piecewise constant (PWC) models with interval uncertainties to approximate the nonlinear characteristics of transistors, encoding DC operating point and transient analyses problems as linear arithmetic constraint satisfaction with promising results.

The key advantage of PWL models lies in their translation to linear arithmetic. On the other hand, the abstraction of transistor characteristics by PWL models can potentially miss DC operating points or transient behaviors, unless the modeling can be made "sound" as defined in Section II. However, a sound model may introduce spurious behaviors that do not exist in the real circuit. To address this issue, Tiwary et al. present a model refinement procedure that constructs a PWL model using a restricted input domain provided by results found from a coarse model.

Thus far, little work has been done to consider the "best" piecewise linear modeling approach for DC and transient analyses of analog circuits. In this paper, we ask the following question: is one modeling approach necessarily better than the other in terms of performance (time taken) vs. accuracy (fewer spurious DC operating points, fewer spurious paths)?

We compare three different PWL modeling approaches, including PWL modeling with simplicial decomposition, the canonical PWL function proposed by Chua et al. [15] and the PWC model used by Tiwary et al. [2] with different modeling parameters. Our comparisons are based on the DC operating point and transient analyses framework of Tiwary et al. [2]. Our comparisons consider the running times, number of SMT solver queries and the precision in terms of spurious results. Our findings are quite surprising: for DC operating point analysis, PWC models are more efficient in terms of performance while providing very little difference in terms of precision. We also present evidence suggesting that PWL models may not be suitable for certain transient analyses.

In the next section, we discuss the device modeling approaches. In section III, we present the setup of the formal DC operating point analysis. Next, we present the formal transient analysis. Finally we discuss the experimental results.

II. Device Modeling Approaches

In this section, we consider the piecewise linear (PWL) modeling of nonlinear analog devices. We discuss the modeling problem based on simulation and introduce the notion of models that are "sound" with respect to data points.

A. Modeling Nonlinear Devices

A device model is a function $\mathbf{y} = \mathbf{F}(\mathbf{x}, \mathbf{p})$, where $\mathbf{y}$ represents the dependent variables, $\mathbf{x}$ the independent variables and $\mathbf{p}$ the device parameters that vary with the fabrication process, voltage and temperature (PVT). For example, a model for a NMOS transistor is

$$I_{DS} = F(V_{GS}, V_{DS}, \mathbf{p}) .$$

In sophisticated device models, such as BSIM4 [1] and PTM [2], F is often complicated and expressed as C (FORTRAN) language subroutines which can be used by SPICE. In order to enable formal verification, we need to abstract F to a simpler, more tractable form.

Device Approximation. Approximations are achieved by means of relational models $R(V_{GS}, V_{DS}, \mathbf{p}, I_{DS})$, that relate possible voltages, parameters values and currents over some domain $\mathbb{D}$. The domain is defined by intervals for V_{GS} and V_{DS} which typically range from 0 to the supply voltage, and some range of parameter uncertainties for $\mathbf{p}$. We require the relation R to be *sound* with respect to the device model.

Definition II.1 (Sound Abstraction)**.** A relational model of a device $R(V_{GS}, V_{DS}, \mathbf{p}, I_{DS})$ is sound with respect to a functional model $I_{DS} = F(V_{GS}, V_{DS}, \mathbf{p})$ if for all $(V_{GS}, V_{DS}, \mathbf{p}) \in \mathbb{D}$,

$$I_{DS} = F(V_{GS}, V_{DS}, \mathbf{p}) \;\Rightarrow\; R(V_{GS}, V_{DS}, \mathbf{p}, I_{DS}) .$$

In other words, the relation R over-approximates the behavior of the device modeled using the function F.

The purpose of piecewise linear modeling of a device is to find a relation R that is sound with respect to some device model such that R is expressible as a linear arithmetic formula. A standard approach for piecewise linear modeling is to find a piecewise linear approximation $\tilde{F}(V_{GS}, V_{DS}, \mathbf{p})$ that minimizes some penalty function

$$\epsilon = \max_{(V_{GS}, V_{DS}, \mathbf{p}) \in \mathbb{D}} |F(V_{GS}, V_{DS}, \mathbf{p}) - \tilde{F}(V_{GS}, V_{DS}, \mathbf{p})| .$$

Given such an $\tilde{F}$, we can obtain a relation R by "bloating" $\tilde{F}$ using the error ϵ:

$$R(V_{GS}, V_{DS}, I_{DS}, \mathbf{p}) : \; I_{DS} \in \tilde{F}(V_{DS}, V_{GS}, \mathbf{p}) + [-\epsilon, \epsilon] .$$

[1]Cf. http://www-device.eecs.berkeley.edu/bsim/.

[2]Cf. http://ptm.asu.edu/.

If $\tilde{F}$ can be expressed in linear arithmetic, then R itself can be expressed in linear arithmetic. In practice, however, the device model F is often not available in a simple closed form, which makes the computation of ϵ difficult. Instead, we use a large number of samples

$$(V_{GS}^{(0)}, V_{DS}^{(0)}, \mathbf{p}^{(0)}, I_{DS}^{(0)}), \; \ldots, \; (V_{GS}^{(N)}, V_{DS}^{(N)}, \mathbf{p}^{(N)}, I_{DS}^{(N)}) ,$$

each consisting of observed voltages, currents and parameter values, to compute the $\tilde{F}$ that minimizes the sample error. We then compute the relation R by using the interval defined by the sample error. The resulting relation R is *sound with respect to samples*, but not necessarily with respect to F. Often, the samples are divided into a smaller *training set* that is used to find $\tilde{F}$, and a large *evaluation set* that is used to compute the error estimate ϵ. If the number of sample points is large and the sampling is uniform, then soundness with respect to samples can be used as a basis for constructing formal models.

Through the rest of the paper, whenever we claim "soundness" of a device model, it refers to soundness with respect to some pre-specified, sufficiently large number of data points.

B. Piecewise Linear Functions

We now discuss various forms of piecewise linear functions and the approximation of nonlinear devices using them.

Consider a domain $\mathbb{D} \subseteq \mathbb{R}^n$. A function $\mathbf{f}(\mathbf{x}) : \mathbb{D} \to \mathbb{R}^m$ is piecewise linear (PWL) if there exists a K-piece partition $S_1, \ldots, S_K$ of $\mathbb{D}$ such that $\mathbf{f}(\mathbf{x})$ can be written as

$$\mathbf{f}(\mathbf{x}) = \begin{cases} \mathbf{a}_1 + \mathbf{B}_1\mathbf{x} & \mathbf{x} \in S_1 \\ \cdots & \cdots \\ \mathbf{a}_K + \mathbf{B}_K\mathbf{x} & \mathbf{x} \in S_K \end{cases} \tag{1}$$

where $\mathbf{a}_i$ are $m \times 1$ vectors, $\mathbf{x}$ is an $n \times 1$ vector, and $\mathbf{B}_i$ are $m \times n$ matrices. We call Equation (1) the conventional form of PWL functions.

Any continuous function $g(\mathbf{x})$ over a bounded domain $\mathbb{D}$ can be approximated to arbitrary accuracy by a PWL function $\tilde{g}(\mathbf{x})$ with a large enough K. However, as K grows, PWL functions become unwieldy.

Simplicial Form. One way to construct a PWL function in conventional form is based on *simplicial decomposition* [39], which subdivides a domain into many simplices $S_1, \ldots, S_K$. Recall that an n dimensional simplex is a polyhedron with $n+1$ vertices. For instance, a 2-simplex is a triangle and a 3-simplex is a tetrahedron. Algorithms for simplicial decomposition of a domain are well-known. If the domain $\mathbb{D}$ is a box, a simplicial decomposition can be constructed in two steps: (1) partition $\mathbb{D}$ into smaller boxes $\mathbb{D}_1, \ldots, \mathbb{D}_k$ by choosing cutpoints along each dimension; (2) subdivide $\mathbb{D}_j$ into simplices. For example, a rectangle yields 2 simplices, and a cuboid yields 5 simplices. Known simplicial decomposition schemes yield $O(n!)$ simplices of n dimensions.

Once we decompose $\mathbb{D}$ into K simplices $S_1, \ldots, S_K$, we assume that for S_i, the linearization is an unknown function $f_i = a_i + \langle \mathbf{b}_i, \mathbf{x} \rangle$, where $\langle\rangle$ denotes inner product. Since S_i has $n+1$ vertices, we evaluate the function $f(\mathbf{x})$ at each vertex and

TABLE I
A CROSS SECTION OF VERIFICATION APPROACHES FOR ANALOG CIRCUITS.

References	Description
Althoff et al. [7]	Verification of Phase Locked Loop.
Frehse et al. [18]	Using Phaver [19] to verify oscillators.
Dang et al. [20]	Verification of $\Delta - \Sigma$ modulator.
Gupta et al. [21]	Using checkmate to verify $\Delta - \Sigma$ modulator [22], [23].
Tiwary et al. [2]	SAT-based D.C analysis, piecewise interval device modeling.
Zaki et al. [6]	Taylor Models Interval Arithmetic [24], [25].
Zaki et al. [3]	DC operating point analysis using nonlinear solvers [10].
Steinhorst et al. [26]	Specification language and model checking by guaranteed integration.
Hartong et al. [4], [27]	Discretization of dynamics and Model checking.
Hartong et al. [28]	Equivalence checking by sampling vector fields at finitely many points.
Little et al. [29]–[31]	Translation to Hybrid Petri Nets and model-checking.
Clarke et al. [32], [33]	Stat. model checking [34] of $\Delta - \Sigma$ modulators.
Singhee et al. [35], [36]	Monte Carlo methods, rare event simulations [37], [38].
Denman et al. [9]	Deriving Laplace transfer functions and verifying using theorem proving.

set up $n+1$ linear equations in terms of a_i and $\mathbf{b}_i$, which yields a unique solution. The resulting f is continuous since the values of f_i and f_j agree at the common vertices of S_i and S_j. A PWL function constructed using simplicial decomposition is said to be a simplicial PWL (SPWL) function.

SPWL functions are practical when the number of inputs is relatively small. For instance, if we assume that the parameters of an NMOS device are fixed, SPWL decomposes the input space in terms of V_{GS} and V_{DS} into triangles. However, if there are many inputs (e.g., V_{GS}, V_{DS} and a number of uncertain transistor model parameters), models based on SPWL can be quite expensive due to the exponential blowup in the number of simplices.

Canonical Form. A continuous PWL function can also be written as:

$$\mathbf{f}(\mathbf{x}) = \mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{i=1}^{\sigma} \mathbf{c}_i \, |\langle \alpha_i, \mathbf{x} \rangle + \beta_i| \tag{2}$$

where $\mathbf{a}$ and $\mathbf{c}_i$ are $m \times 1$ vectors, $\mathbf{x}$ and α_i are $n \times 1$ vectors, B is an $m \times n$ matrix, and β_i is a scalar. Equation (2) is known as the *canonical* form [15], which is more succinct than the conventional form.

We construct a PWL function in canonical form (CPWL) as follows. First, we sample over $\mathbb{D}$ to obtain N samples $\mathbf{x}_i, y_i$, where $1 \leq i \leq N$. Next, we use a gradient descent method that minimizes the error between the output values and the sample points. The gradient descent method is detailed in [15]. Here we present a brief description.

Consider a real-valued function $f(\mathbf{x})$ and a CPWL function

$$\hat{f}(\mathbf{x}) = a + \mathbf{b}\mathbf{x} + \sum_{j=1}^{\sigma} c_j \, |\langle \alpha_j, \mathbf{x} \rangle + \beta_j| \ ,$$

where a, $\mathbf{b}$, c_j, α_j and β_j are unknown coefficients. Given a set of N samples $\{(\mathbf{x}_i, y_i) \mid y_i = f(\mathbf{x}_i)\}$, let

$$z_1 \equiv [a \; b_1 \cdots b_n \; c_1 \cdots c_\sigma]^T \ ,$$
$$z_2 \equiv [\alpha_{1,1} \cdots \alpha_{k,n} \; \beta_1 \cdots \beta_k]^T \ ,$$

and define the L_2-norm error as

$$E(z_1, z_2) \equiv \sum_{i=1}^{N} \left[w^{(i)} \left(\hat{f}(\mathbf{x}_i) - f(\mathbf{x}_i) \right)^2 \right] \ ,$$

where $w^{(i)}$ is the weight of the i-th sample. The L_2-norm error $E(z_1, z_2)$ is minimized by iteratively moving z_2 along the steepest descent direction and computing the local minimum with respect to z_1. When the error reaches a minimum, or is below some threshold, we find an approximation $\hat{f}$.

We simplify the above algorithm as follows. We fix z_2 such that it subdivides the domain $\mathbb{D}$ into hyper-rectangles. Then we compute the local minimum of E, where we get a set of values for z_1. The resulting z_1, along with the pre-selected z_2, leads to a function that generally does not have the minimal error. As shown later, we will "bloat" this function into a sound abstraction. Therefore, instead of getting the CPWL function with minimal error, our concern is more on obtaining a reasonable approximation with low computational effort.

Piecewise Constant Functions. When $\mathbf{B}_1, \ldots, \mathbf{B}_K$ in Equation (1) are set to zero, a useful sub-class of functions is obtained: piecewise constant functions (PWC). They trade off accuracy for computational efficiency.

C. PWL Device Modeling

A PWL device model of a set of samples $\{\mathbf{x}_i, F(\mathbf{x}_i)\}$ is a pair of PWL functions $\tilde{F}_l$ and $\tilde{F}_u$ such that for all i, $\tilde{F}_l(\mathbf{x}_i) \leq F(\mathbf{x}_i) \leq \tilde{F}_u(\mathbf{x}_i)$. Hence, a PWL device model is a relational model that is sound with respect to the samples. We assume that for $\mathbf{x} \in \mathbb{D}$, $F(\mathbf{x})$ can be evaluated. Without loss of generality, we also assume that $\mathbb{D}$ is a box obtained as the Cartesian product of intervals $I_1, \ldots, I_n$ for each x_i of $\mathbf{x}$.

Model Generation. We generate a PWL model for $F(\mathbf{x})$ as follows. First, we construct a PWL function $\tilde{F}(\mathbf{x})$ using the procedures described in the previous section. Then for a set of N samples $\{\mathbf{x}_i, F(\mathbf{x}_i)\}$, which we call the evaluation set, we compute an empirical error estimate

$$\hat{\epsilon} = \max_{1 \leq i \leq N} |F(\mathbf{x}_i) - \tilde{F}(\mathbf{x}_i)| \, .$$

We add the interval $[-\hat{\epsilon}, \hat{\epsilon}]$ to the function $\tilde{f}$ to obtain a relational model that is sound with respect to the samples.

One refinement of this approach computes $\hat{\epsilon}_l$ and $\hat{\epsilon}_u$ that capture the under-approximation and over-approximation errors respectively. Furthermore, $\hat{\epsilon}_l$ and $\hat{\epsilon}_u$ can be computed for each S_i, which provides a more fine-grained error estimation. For CPWL functions, the piecewise estimation is not immediate since the subdivisions are represented implicitly.

The construction of a PWC model is straightforward. Given a partition $S_1, \ldots, S_K$, we simply compute the minimal and maximal values of $F(\mathbf{x})$ for each S_i. This results in a function interval $[\tilde{F}_l, \tilde{F}_u]$ that contains all the samples.

Model Encoding. Finally, we consider the encoding of the models in linear arithmetic. Figure 1 shows the schema for encoding PWC, SPWL and CPWL models. Let $n = |\mathbf{x}|$ be the number of inputs and assume that each component x_i is subdivided into k parts. We define the size of a formula in terms of n and k. A PWC model considers $K = k^n$ boxes. For each box, the size of the formula is $O(n)$. Hence, the size of the encoding is $O(k^n n)$. For an SPWL model, we have $K = O(k^n n!)$, assuming a fixed simplicial decomposition scheme that divides a cube into $n!$ simplices. The size of a formula for each box is also $O(n)$, resulting in a encoding whose size is $O(k^n n!)$. A CPWL model encodes $K = (k+1)n$ boundaries (the absolute value terms in the canonical representation) rather than boxes. Each boundary equation has a size of $O(n)$, yielding an encoding of size $O(kn^2)$.

Thus, CPWL models have the most economical encoding, while SPWL results are potentially the least efficient. However, note that even if two formulae are of the same size, they are not necessarily equivalent in terms of computational effort.

III. Formal DC Operating Point Analysis

The goal of formal DC operating point analysis is to list all DC operating points of a circuit. The standard approach to this problem consists of two steps: [2], [3] (a) Encode the DC operating point condition as constraints, and (b) subdividing the input and output voltages into many regions, query the solvers to find if an operating point can exist inside a given input/output region pair. The nonlinear devices are modeled as described in Section II.

We note that DC operating regions, especially metastable regions are relatively hard to identify using simulation tools like SPICE. A common approach is to perform simulations with the circuit initialized near a potential DC operating point and check if the circuit settles to a nearby DC operating point (see, for example [40]).

Circuit Encoding. The circuit encoding consists of the Kirchhoff's current law (KCL) and the device models. The KCL asserts that the current flowing into a node is equal to the current flowing out. If a node connects to a voltage source or ground, its encoding is unnecessary since the current of a voltage source or ground is unconstrained. PWL device models are generated and encoded in linear arithmetic as discussed in Section II.

Abstraction Refinement. The DC analysis can be performed "monolithically" by a single fine-grained encoding, followed by numerous queries over regions that can "pinpoint" a DC operating point to the required degree of precision. A more efficient top-down approach is suggested by Tiwary et al. [2] wherein the DC operating points are discovered by repeated subdivision much like a branch-and-bound scheme. Initially, large regions are queried for the presence of a DC operating point using a coarse PWL model. If the solver returns a satisfiable answer, then the regions are subdivided and refined PWL models are fitted to these regions.

Spurious Region. We call a region *spurious* if it is reported by the analysis, but does not actually contain operating points. Spurious regions are produced by the sound abstraction of device models which over-approximates the behavior of devices. Consider the inverter in Figure 2 with its input fixed to $0.5V$. The output can vary between $0.4V$ and $0.6V$ due to the abstraction (in contrast to $0.5V$ in reality). Suppose the transistors are linearized on the regions $0.35 \leq V_{out} \leq 0.45$, $0.45 \leq V_{out} \leq 0.55$ and $0.55 \leq V_{out} \leq 0.65$. Then the regions $[0.35, 0.45]$ and $[0.55, 0.65]$ become spurious. A finer abstraction may lead to fewer spurious regions. But it also results in a more complicated model.

IV. Formal Transient Analysis

The abstraction of nonlinear devices also enables formal transient analysis. Formal transient analysis deals with reachability problems, i.e, given an initial condition, whether the circuit output can reach values in some range. A simple approach proposed by Tiwary et al. [2], is to generate an approximate transition relation by encoding the change in voltages and currents across capacitors and inductors in the circuit. The resulting change is approximated by an Euler step. While such a transient analysis is a poor alternative to the more sophisticated approach adopted by linear hybrid systems based approaches [7], [41], it allows us to encode the approximate reachability by means of a BMC formula. This can be a potentially faster approach to exploring all possible behaviors for a bounded time interval.

We employ the transient analysis scheme as yet another evaluation method for comparing the various PWL models considered in Section II. However, we note that the Euler step can be a large over-approximation unless the time step is small. However, a small time step also means that the depth of the BMC encoding needs to be larger to perform time bounded reachability up to the same time interval.

V. Experimental Evaluation

In this section, we compare the various device modeling approaches, PWC, SPWL and CPWL. We apply them to formal DC operating point and transient analyses, as described in Section III and Section IV. We implement (using the Python programming language) the various modeling approaches, DC and transient analyses. Our program processes the input circuit as a net list and builds a linear arithmetic formula. We use the Z3 SMT solver to check the satisfiability of these formulae

$$\begin{aligned} S_1(\mathbf{x}) &\Rightarrow y \in f_1(\mathbf{x}) + [-\epsilon_1, \epsilon_1] \\ &\cdots \\ S_K(\mathbf{x}) &\Rightarrow y \in f_K(\mathbf{x}) + [-\epsilon_K, \epsilon_K] \end{aligned}$$

$$\begin{gathered} y \in \mathbf{a} + B\mathbf{x} + \textstyle\sum_{i=1}^{K} c_i r_i + [-\epsilon, \epsilon] \\ \langle \alpha_1, \mathbf{x}_1 \rangle + \beta_1 \geq 0 \Rightarrow r_1 = \langle \alpha_1, \mathbf{x}_1 \rangle + \beta_1 \\ \langle \alpha_1, \mathbf{x}_1 \rangle + \beta_1 < 0 \Rightarrow r_1 = -(\langle \alpha_1, \mathbf{x}_1 \rangle + \beta_1) \\ \cdots \\ \langle \alpha_K, \mathbf{x}_K \rangle + \beta_K \geq 0 \Rightarrow r_K = \langle \alpha_K, \mathbf{x}_K \rangle + \beta_K \\ \langle \alpha_K, \mathbf{x}_K \rangle + \beta_K < 0 \Rightarrow r_K = -(\langle \alpha_K, \mathbf{x}_K \rangle + \beta_K) \end{gathered}$$

Fig. 1. **(Left)** Schema for encoding SPWL and PWC models; and **(Right)** CPWL encoding.

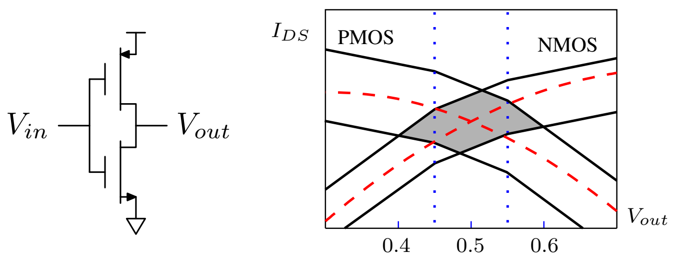


Fig. 2. An inverter and its I/O characteristics. Straight lines show the PWL models of the two devices. Dashed lines are the SPICE models. Shaded region illustrates the approximation due to abstraction. Note that there is only one operating point in the region.

under different input/output intervals. All experiments were run using a Ubuntu 11.10 Desktop on a Quad-core 2.8 GHz machine with 9 GB memory.

We list the benchmarks with brief descriptions in Table II. The letters in the second column refer to the types of devices. M stands for MOSFET transistors, R for linear resistors, C for linear capacitors and L for linear inductors. The numbers count how many devices there are in each type.

The first four benchmarks are ring oscillators with different numbers of stages. The benchmarks starting with "evenosc" are even-stage oscillators from [1]. Their schematic is shown in Figure 3(a). The suffixes "lcbr" and "scbr" denote oscillator benchmarks with known bugs: the oscillators fail due to incorrect transistor sizing [1]. The benchmark "sqwavegen" is a square-wave generator based on a CMOS Schmitt trigger. The "lctankvco" is a voltage-controlled oscillator that uses the inductors and capacitors as the source of oscillation and the cross-coupled pair of transistors as negative resistors to compensate the energy dissipation in the inductor resistance. The schematics of "sqwavegen" and "lctankvco" are shown in Figure 3(b) and 3(c), respectively.

TABLE II
BENCHMARKS FOR DC AND TRANSIENT EXPERIMENTS.

Name	Size	Description
ringosc3s	$6M$	Ring oscillators
ringosc5s	$10M$	
ringosc7s	$14M$	
ringosc9s	$18M$	
evenosclcbr	$16M$	Even-stage oscillators
evenoscscbr	$16M$	
evenoscncbr	$16M$	
sqwavegen	$6M, 1R, 1C$	A square-wave generator
lctankvco	$4M, 2R, 2C, 2L$	An LC-tank VCO

A. Formal DC Operating Point Analysis

In this part, our goal is to compare the performance of different device modeling approaches in terms of accuracy with respect to SPICE simulation, the number of SMT queries and the running time. The setup is as follows: we fix the device parameters and apply the abstraction refinement described in Section III. The initial number of subdivisions is set to 2 along each dimension. Each refinement step further subdivides each region by splitting each axis into 2 pieces. The refinement process is applied recursively to further subdivide regions that are deemed to contain potential operating points. This process stops after a depth-cutoff that is set to 3 for our experiments.

We carry out our experiments using PWL models with k subdivisions along each dimension, where $k = 1, 2, 4, 6, 8, 16$. We prefix k to denote the specific approach. For instance, 4-PWC stands for PWC models with $k = 4$. We omit 1-CPWL because it does not fit into the algorithm for generating PWL functions in canonical form [15].

We first report the number of regions that may contain operating points by each approach in Table III. The number of operating regions confirmed by SPICE is shown in the last column. The regions found by various PWL models that are not confirmed by SPICE are spurious. In Table III, the number of spurious regions is simply the total number of reported regions minus the number of real operating points. The SPICE-based DC operating point discovery is a trial-and-error process, since the operating points may be metastable.

Accuracy. We compare accuracy in terms of the number of spurious regions in Table III. Observe that SPWL and CPWL are only marginally better than PWC. For the ring oscillator examples, none of the methods reports spurious regions. For the rest of the examples, the number of regions is generally more than twice larger than the number SPICE confirmed. Not surprisingly, we observe that the spurious regions are neighbors to the confirmed regions. We also observe that with more subdivisions, the three approaches tend to get similar results. This shows that the error in the approaches are negligible making their predictions very similar.

SMT Queries. Next, we compare the number of SMT queries in Table IV. The number of SMT queries is a proxy for the number of regions where DC queries occur for a potential operating point. Here, we see that PWC models consistently require more queries than SPWL and CPWL. That is because PWC models over-approximate the underlying device behavior the most, and therefore produce a lot of false positives that are subsequently pruned by refinement. Also, we see that SPWL

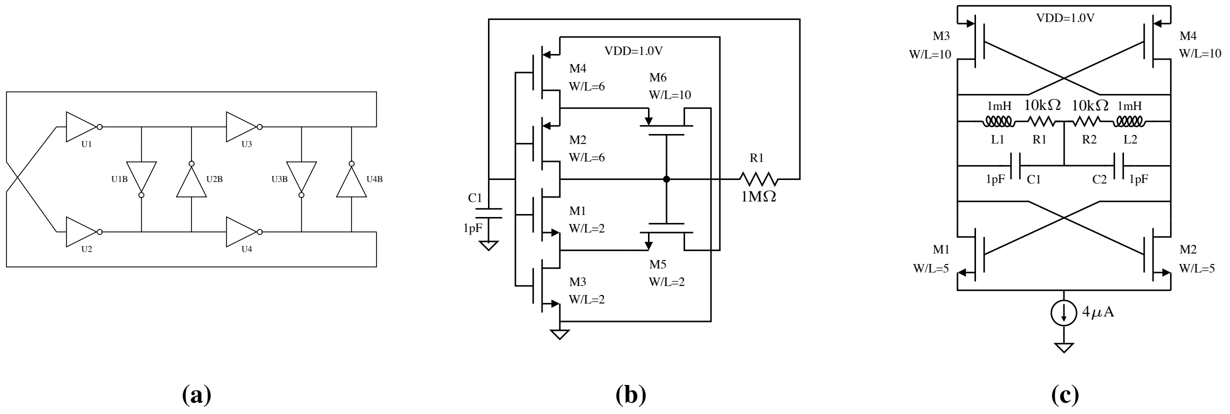


Fig. 3. Circuit diagrams for **(a)** an even-stage oscillator [1], **(b)** a square-wave generator, and **(c)** a voltage controlled oscillator

TABLE III
THE NUMBER OF SPURIOUS REGIONS, WHICH MAY NOT CONTAIN OPERATING POINTS, REPORTED BY EACH APPROACH. THE LAST COLUMN SHOWS THE NUMBER OF REAL OPERATING POINTS OBTAINED FROM SPICE SIMULATION. THE NUMBERS IN THE SECOND ROW REFER TO THE SUBDIVISION OF THE CORRESPONDING APPROACH ALONG EACH DIMENSION. "TO" MEANS MORE THAN 500 SECONDS.

	PWC						SPWL						CPWL					SPICE
	1	2	4	6	8	16	1	2	4	6	8	16	2	4	6	8	16	
ringosc3s									– 0 –									1
ringosc5s									– 0 –									1
ringosc7s									– 0 –									1
ringosc9s	TO						– 0 –					TO			– 0 –			1
evenosclcbr	14	4	4	4	4	0	4	4	0	0	TO	TO	4	4	4	4	TO	3
evenoscscbr	10	10	6	6	6	6	6	6	6	6	TO	TO	6	6	2	2	TO	3
evenoscncbr	22	12	12	12	12	12	12	12	12	TO	TO	TO	2	TO	2	2	TO	1
sqwavegen	7	5	4	3	3	1	2	1	3	3	3	1	3	2	2	1	1	1
lctankvco	9	9	5	5	3	3	1	3	2	2	2	2	5	5	4	3	3	1

TABLE IV
THE NUMBER OF SMT QUERIES FOR EACH APPROACH. ">" MEANS TIME-OUT WITH A 500 SECONDS LIMIT, AND THE FOLLOWING VALUE IS THE NUMBER OF QUERIES AT TIME-OUT.

	PWC						SPWL						CPWL				
	1	2	4	6	8	16	1	2	4	6	8	16	2	4	6	8	16
ringosc3s	97	13	7	7	7	7	33	13	7	7	7	7	13	7	7	7	7
ringosc5s	833	27	7	7	7	7	149	27	7	7	7	7	27	7	7	7	7
ringosc7s	7937	63	7	7	7	7	653	63	7	7	7	7	63	7	7	7	7
ringosc9s	>43291	157	7	7	7	7	2841	157	7	7	7	>5	157	7	7	7	7
evenosclcbr	929	91	35	27	27	27	213	71	35	27	>26	>3	63	35	27	27	>25
evenoscscbr	801	75	63	55	47	47	221	67	63	55	>34	>3	59	55	39	31	>17
evenoscncbr	1185	119	75	63	55	55	177	79	63	>33	>14	>3	83	>67	63	55	>10
sqwavegen	161	55	25	27	19	15	57	35	25	23	17	15	51	25	27	19	13
lctankvco	149	71	29	27	17	13	59	53	15	19	17	13	71	21	19	17	13

is a slightly better than CPWL. Again, the approaches become quite similar as the number of subdivisions increases.

Running Time. Finally, we consider the running time of each approach with different subdivisions in Table V. It is obvious that PWC models are superior to the remaining models, even though they require more queries to the SMT solver. A likely explanation is that each SMT query from the PWC approach is simpler and far easier to solve than the corresponding queries from the SPWL and CPWL approaches. The results for CPWL models are interesting. First, the running times do not necessarily grow with decreased granularity. Even for a single SMT query, the average solving time does not increase as fast as the other two approaches. A possible explanation is that the complexity of CPWL models grows linearly with the number of subdivisions, unlike the other two approaches, whose models grow exponentially. On the other hand, even the simplest CPWL models take considerably more time than PWC and SPWL counterparts. It suggests that the CPWL encoding is difficult for SMT solvers.

In summary, as the number of subdivisions in the models increases, PWC outperforms SPWL and CPWL in the given set of benchmarks. The running times of PWC and SPWL grow as the granularity decreases. On the other hand, the running times of CPWL are less predictable.

TABLE V
RUNNING TIME FOR EACH APPROACH WITH A 500 SECONDS TIME-OUT.

	PWC						SPWL						CPWL				
	1	2	4	6	8	16	1	2	4	6	8	16	2	4	6	8	16
ringosc3s	<0.1	0.1	0.1	0.4	0.9	11	0.4	0.4	1.6	15	34	339	0.5	1.5	8.6	6.7	33
ringosc5s	2.8	0.3	0.2	1.5	1.5	18	2.8	1.4	3.3	17	42	365	15	4.4	8.3	13	54
ringosc7s	75	0.6	0.2	0.8	2	26	17	4.2	4.3	15	30	376	17	5.1	19	16	104
ringosc9s	>500	2.1	0.3	1	2.6	33	95	16	6.8	18	44	>500	61	17	36	82	247
evenosclcbr	7.4	1.1	1.3	3.2	9.2	146	17	29	152	368	>500	>500	59	158	111	171	>500
evenoscscbr	6.5	1	2.2	7	16	277	17	29	169	406	>500	>500	85	300	371	319	>500
evenoscncbr	9.7	1.7	3.5	11	29	444	22	73	246	>500	>500	>500	199	>500	343	438	>500
sqwavegen	0.2	0.5	0.5	1.6	2.7	27	0.9	1.1	3.5	9.2	16	120	0.8	1.1	2.6	4	19
lctankvco	0.3	0.7	0.6	2	2.7	35	0.6	1.4	2	8.4	15	125	0.9	0.8	1.8	2.7	22

B. Formal Transient Analysis

We use the ring oscillator benchmarks to compare the performance of the three approaches, PWC, SPWL and CPWL, on formal transient analysis. The setup is as follows: for each method, we perform time-bounded reachability queries for different time frames, and compare the accuracy of the results relative to SPICE simulations, which report a single concrete voltage value at each time interval. A reachability query checks whether the output can reach a certain interval in a specified time frame. We set the initial output voltage to $1.0V$ and subdivide the range of the output voltages into ten intervals, each of which is queried individually. We use backward Euler integration to solve the transient behavior of active devices. The time step is fixed to a value that is small enough to obtain accurate integration results.

TABLE VI
REACHABLE INTERVALS (OVER-APPROXIMATIONS) FOUND BY VARIOUS APPROACHES FOR DEPTHS 1, 5 AND 10 STEPS OF TRANSIENT ANALYSIS FOR THE THREE-STAGE RING OSCILLATOR. THE REACHABLE SETS ARE INDEPENDENT OF TRANSISTOR SIZING. RUNNING TIMES ARE LISTED IN TABLE VII.

	Reachable Interval	Approach
1-step	[0.9,1.0]	all approaches
5-step	[0.7,1.0]	16-PWC
	[0.6,1.0]	remaining approaches
10-step	[0.4,1.0]	4-PWC, 6-PWC and 8-PWC
	[0.3,1.0]	remaining approaches

We simulate for one time frame, five time frames and ten time frames respectively. The results are shown in Table VI and Table VII. Barring timeouts, the three approaches produce almost identical reachability results.

First, let us observe that a better model is helpful in getting a more accurate reachable interval. For instance, 4-PWC reports a reachable interval of $[0.4, 1.0]$ at the tenth time frame, while 2-PWC concludes a larger reachable interval: $[0.3, 1.0]$. However, notice that the reachable intervals are generally too over-approximate compared to the SPICE simulations which report a single concrete value at each time step. Therefore, the approximations seem to be too coarse to provide useful reachability information. On the other hand, increasing the number of subdivisions makes the computation intractable (Table VII).

In terms of running time, there are many time-outs for each approach. This may seem surprising since the benchmarks are small and the number of time frames is not large. We suspect that the PWL encoding forces the SMT solver to explore a large set of transistor mode combinations, wherein each subdivision in the PWL represents a transistor mode. The number of such mode combinations increases exponentially as the unrolling depth is increased. The observations suggest that a simple BMC-style encoding of transient analysis may be suboptimal in terms of performance and accuracy.

VI. CONCLUSION

To summarize this paper, we compare the applicability of three device modeling approaches, PWC, SPWL and CPWL, to the formal DC operating point and formal transient analysis. We find that PWC is the most suitable approach for operating point analysis. Both SPWL and CPWL generate more complicated models. The benefits from those models, for instance, fewer spurious regions and fewer SMT queries, do not compensate the extra cost in terms of solving time.

On the other hand, none of the approaches performs well for transient analysis with the described simulation scheme in the selected benchmark set. The results suggest that with a sound abstraction of device models, the simple BMC-style unrolling does not work well.

In the future, it is interesting to identify whether a region contains a stable or metastable operating point. Also, we can utilize the unsatisfiable core of SMT queries, which can potentially rule out more than one region each time. Techniques from unsatisfiablity solvers, such as iSAT [10], can also be applied to the DC operating points analysis.

REFERENCES

[1] K. D. Jones, J. Kim, and V. Konrad, "Some "real world" problems in the analog and mixed signal domains," in *Proceedings of Designing Correct Circuits*, 2008.

[2] S. K. Tiwary, A. Gupta, J. R. Phillips, C. Pinello, and R. Zlatanovici, "First steps towards SAT-based formal analog verification," in *ICCAD*, 2009, pp. 1–8.

[3] M. H. Zaki, I. M. Mitchell, and M. R. Greenstreet, "DC operating point analysis - a formal approach," in *Proceedings of Formal Verification of Analog Circuits (FAC)*, 2009.

[4] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127.

[5] M. Freibothe, J. Döge, T. Coym, S. Ludwig, B. Straube, and E. Kock, "Verification-oriented behavioral modeling of nonlinear analog parts of mixed-signal circuits," in *Advances in Design and Specification Languages for Embedded Systems*, 2007, pp. 37–51.

TABLE VII
RUNNING TIME OF 1-STEP, 5-STEP AND 10-STEP TRANSIENT ANALYSIS.

		PWC					SPWL					CPWL				
		2	4	6	8	16	2	4	6	8	16	2	4	6	8	16
1-step	ringosc3s	0.1	0.1	0.2	0.3	2.4	0.1	0.3	0.5	1.1	5.6	0.1	0.2	0.3	0.5	2.8
	ringosc5s	0.1	0.2	0.3	0.8	7.1	0.2	0.6	1.4	3.1	18	0.1	0.4	1	1.3	7.8
	ringosc7s	0.1	0.2	0.6	1.2	11	0.3	0.8	2	3.8	27	0.2	0.5	0.8	1.8	13
	ringosc9s	0.2	0.4	0.8	1.8	17	0.5	1.4	3.2	6.4	47	0.3	0.6	1.3	2.1	17
5-step	ringosc3s	0.9	4.7	9	13	80	10	53	191	372	>500	41	44	92	106	430
	ringosc5s	1.9	16	40	59	430	18	221	>500	>500	>500	>500	95	258	317	>500
	ringosc7s	4.3	34	94	162	>500	31	306	>500	>500	>500	>500	161	384	>500	>500
	ringosc9s	3.9	57	146	200	>500	32	>500	>500	>500	>500	>500	478	>500	>500	>500
10-step	ringosc3s	6.7	50	88	185	>500	200	>500				>500				
	ringosc5s	44	>500				372	>500				>500				
	ringosc7s	23	>500				>500	>500				>500				
	ringosc9s	45	>500				>500	>500				>500				

[6] M. H. Zaki, G. Al-Sammane, S. Tahar, and G. Bois, "Combining symbolic simulation and interval arithmetic for verification of AMS designs," in *FMCAD*, 2007, pp. 207–215.

[7] M. Althoff, A. Rajhans, B. Krogh, S. Yaldiz, X. Li, and L. Pileggi, "Formal verification of phase-locked loops using reachability analysis and continuation," in *ICCAD*, 2011, pp. 659–666.

[8] S. Little, D. Walter, K. Jones, C. J. Myers, and A. Sen, "Analog/mixed-signal circuit verification using models generated from simulation traces," *International Journal of Foundations of Computer Science*, vol. 21, no. 2, pp. 191–210, 2010.

[9] W. Denman, B. Akbarpour, S. Tahar, M. H. Zaki, and L. C. Paulson, "Formal verification of analog designs using MetiTarski," in *FMCAD*, 2009, pp. 93–100.

[10] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige, "Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure," *Journal on Satisfiability, Boolean Modeling and Computation, Special Issue on SAT/CP Integration*, pp. 209–236, 2007.

[11] S. Gao, M. K. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E. M. Clarke, "Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems," in *FMCAD*, 2010, pp. 81–89.

[12] M. K. Ganai and F. Ivancic, "Efficient decision procedure for non-linear arithmetic constraints using CORDIC," in *Proceedings of the Formal Methods in Computer Aided Design*, 2009, pp. 61–68.

[13] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. L. Sangiovanni-Vincentelli, "CalCS: SMT solving for non-linear convex constraints," in *FMCAD*, 2010, pp. 71–79.

[14] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *(TACAS)*, 2008, pp. 337–340.

[15] L. O. Chua and A.-C. Deng, "Canonical piecewise-linear modeling," *IEEE Transactions on Circuits and Systems*, no. 5, pp. 511–525, 1986.

[16] M.-J. Chien and E. S. Kuh, "Solving nonlinear resistive networks using piecewise-linear analysis and simplicial subdivision," *IEEE Transactions on Circuits and Systems*, vol. 24, no. 6, pp. 305–317, 1977.

[17] V. B. Rao, D. V. Overhauser, T. N. Trick, and I. N. Hajj, *Switch-Level Timing Simulation of MOS VLSI Circuits*. Kluwer Academic Publishers, 1989.

[18] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *DATE*, 2006, pp. 257–262.

[19] G. Frehse, "Phaver: Algorithmic verfication of hybrid systems past hytech," in *HSCC*, 2005, pp. 258–273.

[20] T. Dang, A. Donze, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid systems techniques," in *FMCAD*, 2004.

[21] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *ICCAD*, 2004, pp. 210–217.

[22] B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutiman, "Modeling and verification of hybrid dynamical system using checkmate," in *ADPM*, 2000.

[23] A. Chutiman and B. Krogh, "Computing polyhedral approximations to flow pipes for dynamic systems," in *Proceedings of IEEE CDC*, 1998.

[24] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. SIAM, 2009.

[25] M. Berz and K. Makino, "Performance of taylor model methods for validated integration of odes," vol. 3732, pp. 69–74, 2005.

[26] S. Steinhorst and L. Hedrich, "Model checking analog systems using an analog specification language," in *DATE*, 2008, pp. 324–329.

[27] W. Hartong, L. Hedrich, and E. Barke, "Model checking algorithms for analog verification," in *DAC*, 2002, pp. 542–547.

[28] W. Hartong, K. Klausen, and L. Hedrich, "Formal verification of non-linear analog systems: Approaches to model and equivalence checking," in *Advanced Formal Verification*, R. Drechsler, Ed. Kluwer, 2004, pp. 205–245.

[29] S. Little, D. Walter, N. Seegmiller, C. Myers, and T. Yoneda, "Verification of analog and mixed-signal circuits using timed hybrid Petri nets," in *Automated Technology for Verification and Analysis*, 2004, pp. 426–440.

[30] S. Little, N. Seegmeller, D. Walter, C. Myers, and T. Yoneda, "Verification of analog/mixed-signal circuits using labeled hybrid Petri nets," in *ICCAD*, 2006, pp. 275–282.

[31] D. Walter, S. Little, C. J. Myers, N. Seegmiller, and T. Yoneda, "Verification of analog/mixed-signal circuits using symbolic methods," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2223–2235, 2008.

[32] E. M. Clarke, A. Donze, and A. Legay, "Statistical model checking of analog mixed-signal circuits with an application to a thrid order δ-σ modulator," in *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, 2009, pp. 149–163.

[33] E. M. Clarke and P. Zuliani, "Statistical model checking for cyber-physical systems," in *Automated Technology for Verification and Analysis*, 2011, pp. 1–12.

[34] H. L. S. Younes and R. G. Simmons, "Statistical probabilistic model checking with a focus on timed-bounded properties," *Information and Computation*, vol. 204, no. 9, pp. 1368–1409, 2006.

[35] A. Singhee and R. A. Rutenbar, "From finance to flip flops: A study of fast quasi-monte carlo methods from computational finance applied to statistical circuit analysis," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, 2007, pp. 685–692.

[36] A. Singhee, S. Singhal, and R. A. Rutenbar, "Practical, fast Monte Carlo statistical static timing analysis: Why and how," in *ICCAD*, 2008, pp. 190–195.

[37] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo Method*. Wiley Series in Probability and Mathematical Statistics, 2008.

[38] ——, *The Cross-entropy Method: An Unified Approach to Combinatorial Optimization, Monte-Carlo Simlation and Machine Learning*. Springer-Verlag, 2004.

[39] A. Hatcher, *Algebraic Topology*. Cambridge university Press, 2002.

[40] P. Varma, B. S. Panwar, and K. N. Ramganesh, "Cutting metastability using aperture transformation," *IEEE Transactions on Computers*, vol. 53, pp. 1200–1204, 2004.

[41] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 379–395.

Forward and Backward : Bounded Model Checking of Linear Hybrid Automata From Two Directions

Yang Yang*, Lei Bu*†, and Xuandong Li*

***Abstract*—Instead of encoding the bounded state space of a linear hybrid automaton(LHA) in given threshold k into SMT formulas then solving them by SMT solvers, the authors proposed a different approach to handle the bounded reachability verification(BMC) of LHA in the previous work. First, the reachability specification along one abstract path in LHA can be checked by linear programming (LP). Then, all the abstract paths under the threshold can be checked one by one by depth-first-search (DFS) traversing. This approach was implemented in a prototype tool BACH, and the experiment result shows it is efficient.**

As BACH uses DFS to traverse the bounded state space, clearly, if DFS traverses more quickly, the BMC can be finished more efficiently. Nevertheless, in many cases, the path segments which make the system infeasible are hidden "deeply" in the model or have many entry points which makes the DFS difficult to find them or has to traverse them many times. This burdens the DFS-style BMC approach a lot.

To handle this problem, in this paper, the authors propose a backward-DFS BMC approach for LHA. First, reverse the graph structure of LHA. Then, conduct the DFS-style BMC on the reversed LHA. In this way, the "deep" path segments in the forward-DFS can be found very quickly to prune the DFS tree which is needed to be traversed. This backward-DFS approach is implemented into BACH. The experiment shows the performance of BACH is optimized significantly to handle large cases.

I. Introduction

Reachability verification of Linear Hybrid Automata (LHA) [1] is a very difficult and important problem. Currently, researchers always try to handle this problem in two ways:

Classical Model Checking(CMC)[4]: Compute the complete state space by methods like polyhedra computation, like HYTECH[6] and PHAVer[7]. First of all, the classical reachability verification problem of LHA is proven to be undecidable[2], [3]. Furthermore, these methods are very complex and sensitive to the number of variables. Thus, they do not scale well to the size of practical problems.

Bounded Model Checking(BMC)[5]: Encoding the bounded reachability problem into the satisfiability problem of a boolean combination of propositional variables and linear mathematical constraints, which can be solved by SMT solvers[8], [9]. As this technique requires to encode the bounded problem space firstly, when the system size or the given threshold is large, the object problem will be very huge, which restricts the size of the problem that can be solved.

Both classical and bounded model checking are feeding the (partly) complete state space to the underlying solver at one time which suffers from the state explosion problem and restricts the problem size that can be solved. Study[13] proposed a linear programming (LP)-based approach to check one abstract path at one time to find whether there exists a behavior of LHA along this path and satisfy the given reachability specification. Study[15] extended this approach by using forward depth-first-search (F-DFS) to traverse on the graph structure of LHA to enumerate and check all the abstract paths with length no longer than the threshold one by one to answer the bounded reachability problem. Furthermore, when the DFS is finished before touching the bound, this approach can prove the given specification is not satisfied in general, not only in the given bound. A prototype tool BACH[15], [16] was implemented based on this idea. The experiments show that BACH is efficient in many cases.

Clearly, if DFS traverses more quickly, the BMC can be finished more efficiently. In BACH, once a path is found to be infeasible, the DFS will ask the underlying LP solver to locate the path segment which makes this path infeasible and backtrack to the path segment to prune the behavior tree which is needed to be traversed [17]. Nevertheless, in many cases, the path segments which make the path infeasible are hidden "deeply" or have many entry points in the graph structure which makes the DFS difficult to find these path segments or has to traverse them time and time again. This asks the DFS to traverse a lot of obviously infeasible paths, which makes the backtracking strategy inefficient in many cases and burdens the DFS-style BMC a lot.

To handle this problem, in this paper, the authors propose a backward-DFS BMC (B-DFS)[10], [11] approach to complement the classical F-DFS. First, reverse the graph structure of LHA. Mark the original target location as initial location, and mark the original initial location as target. Then, conduct the F-DFS BMC on the reversed LHA. As these path segments are "deep" in the original graph structure, clearly, if the DFS is conducted in a backward way, the path segments will be "shallow" in the reversed graph and can be found more quickly. Furthermore, for those path segments which have many entry points, the abstract paths with these path segments as suffix will be pruned easily in the reversed LHA to shrink the size of the DFS tree which is needed to be traversed.

This B-DFS BMC idea is implemented into BACH as a complement to the F-DFS BMC. Once a LHA and a reachability specification is given, BACH will start two threads, one conducts the F-DFS BMC on the original LHA and the other conducts the B-DFS BMC on the reversed model. The procedure terminates when any of these two threads finish. We conduct a series of case studies on the new BACH, and

*State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, P.R.China, 210046 Email: yangyang@seg.nju.edu.cn,{bulei, lxd}@nju.edu.cn
† Corresponding author.

compare it with both state-of-the-art classical model checker PHAVer and SMT solver MathSAT. The experiment results show that:

- By B-DFS, the state space needed to search and verify is pruned substantially. Therefore, BACH outperforms the other competitors significantly.
- In many cases, the DFS is finished before touching the bound. Then, in this situation, BACH can prove the given specification is not satisfiable in general, not only in the given bound, which is incapable for other BMC checkers.

II. The Underlying Techniques

A. Forward-DFS Approach

Now, let's recall the F-DFS BMC approach given in study [15]. The main idea is to traverse all the abstract paths with length shorten than or equal to the bound by DFS on the graph structure of the LHA. Whenever an abstract path ρ, which contains the target location, is found, a decision procedure will be called to check whether the LHA has a feasible behavior according with ρ. The decision procedure will encode all the syntax elements, i.e., invariants, guards and e.t.c., in ρ into a linear constraint set $\mathbb{R}$ as guided by [13] and [14]. In this manner, the reachability problem of ρ is transformed into the feasibility of $\mathbb{R}$ which can be answered by an LP solver very efficiently.

It is obvious that the performance of the F-DFS-based BMC depends on the performance of the DFS algorithm. If the DFS can be finished more quickly, the BMC can be finished more efficiently. Therefore, we gave an optimization technique to generate unsatisfiable core from the proven infeasible paths to tailor the state space needed to be traversed[17]. Once ρ is proved to be infeasible by the underlying LP solver, the solver will provide a infeasible irreducible set (IIS) [18] of $\mathbb{R}$, which is a minimal set of constraints in $\mathbb{R}$ that makes $\mathbb{R}$ infeasible. Then the constraint set $\mathbb{R}'$ that IIS located can be mapped back to a path segment ρ' in the path ρ, which means ρ' is the infeasible core in the path that makes ρ infeasible. Since removing any constraint in $\mathbb{R}'$ will make it feasible, so the DFS will backtrack to the location preceding the last location in ρ'.

Take the sample LHA in Fig.1 for example. We want to check whether location v_6 is reachable within bound 20. Suppose the current visiting path is $\rho = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$ which is proved to be infeasible by an LP solver, and the infeasible path segment located by IIS is $\rho' = \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle$, so the location that DFS will backtrack to is v_3 instead of v_5. Then the DFS will begin to search the next branch under $\rho'' = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle$, which is $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_9]{} \langle v_1\rangle \xrightarrow[e_8]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$. As we can see, the subtree beneath $\rho'' \xrightarrow[e_3]{} \langle v_4\rangle$ is pruned completely to speed up the DFS.

Nevertheless, let's look at the following path $\rho_1 = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_9]{} \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$. ρ_1 is a graphically correct path beneath ρ'' and contains target location v_6. Therefore, the F-DFS BMC will ask the underlying

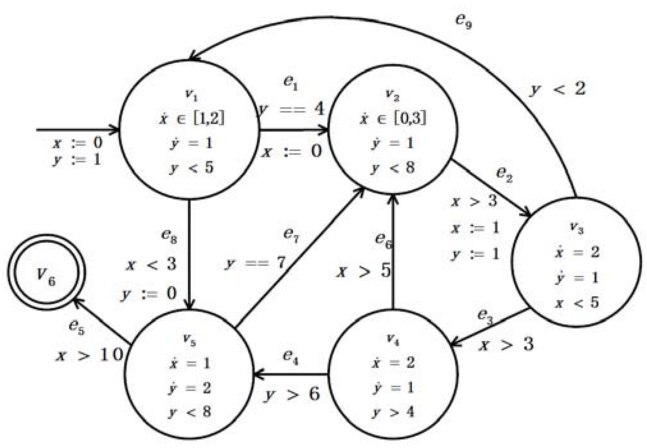


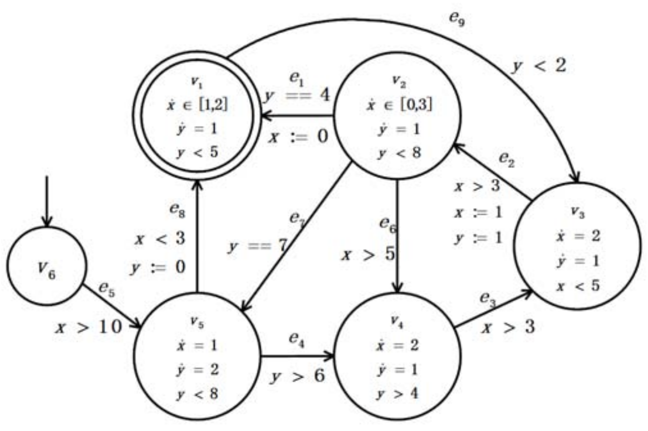
Fig. 1. Sample Automaton

decision procedure to check whether ρ_1 is feasible or not. Indeed, as ρ_1 contains ρ', it can be proved to be infeasible at once. Then DFS will backtrack to the second v_3 in ρ_1. But, as the bound 20 is not reached yet, the DFS will traverse the loop $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle$ for the third time and then ρ' again, i.e., traverse a new path $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_9]{} \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_9]{} \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$. In this manner, lots of time will be wasted on DFS to traverse these paths which are doomed to be infeasible. Obviously, if there is a method to prune the DFS tree and avoid the visiting of such paths, the DFS-BMC will be much more efficient.

B. Backward-DFS Approach

As discussed above, in many times, the path segments which make the path infeasible are hidden quite "deeply" or could have many entries in the graph structure which makes the F-DFS difficult to find them or has to traverse them lots of times. To solve this problem, in this paper we propose to conduct the DFS search in a backward way(B-DFS BMC): First, reverse the transition relation of the LHA model. For each transition $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle$ in LHA H, there will be a transition $\langle v_2\rangle \xrightarrow[e_1]{} \langle v_1\rangle$ in the reversed LHA $\neg H$. Then mark the original target location as initial location, and mark the original initial location as target. For example, Fig.2 is the reversed model of the LHA given in Fig.1.

Now, we can conduct the F-DFS BMC on the reversed LHA, which means searching for a reversed path from the original target location to the initial location. Once a reversed path is found, we will ask the LP solver to check the accordingly path in the original model and locate the infeasible path segments as well. As many infeasible path segments are "deep" in the original graph structure, if the DFS is conducted in a backward way, these path segments will be "shallow" in the reversed structure and can be found more quickly. Furthermore, for those path segments which have many entry points, all the paths containing those path segments as "suffix" will be reversed, then those infeasible path segments will become "prefix" now. As a result, these paths can be pruned easily in the reversed LHA to shrink the size of the DFS tree which is needed to be traversed.



e_9
$y < 2$
v_1 $\dot{x} \in [1,2]$ $\dot{y} = 1$ $y < 5$
e_1 $y == 4$ $x := 0$
v_2 $\dot{x} \in [0,3]$ $\dot{y} = 1$ $y < 8$
e_2 $x > 3$ $x := 1$ $y := 1$
v_3 $\dot{x} = 2$ $\dot{y} = 1$ $x < 5$
e_8 $x < 3$ $y := 0$
e_7 $y == 7$
e_6 $x > 5$
v_6
e_5 $x > 10$
v_5 $\dot{x} = 1$ $\dot{y} = 2$ $y < 8$
e_4 $y > 6$
v_4 $\dot{x} = 2$ $\dot{y} = 1$ $y > 4$
e_3 $x > 3$

Fig. 2. Reversed Sample Automaton

Now, let's look at the previous example again (whether v_6 is reachable within bound 20 in Fig.1). On the reversed model, Fig.2, the first path that is traversed and solved is: $\neg\rho_1 = \langle v_6\rangle \xrightarrow[e_5]{} \langle v_5\rangle \xrightarrow[e_4]{} \langle v_4\rangle \xrightarrow[e_3]{} \langle v_3\rangle \xrightarrow[e_2]{} \langle v_2\rangle \xrightarrow[e_1]{} \langle v_1\rangle$. The accordingly path in the original model is $\rho_1 = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$. Therefore, ρ_1 is given to the underlying LP solver. The LP solver can tell that ρ_1 is infeasible and $\rho_1' = \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle$ is the infeasible core. The according infeasible core in $\neg\rho_1$ is $\neg\rho_1' = \langle v_4\rangle \xrightarrow[e_3]{} \langle v_3\rangle \xrightarrow[e_2]{} \langle v_2\rangle$. So, the location which is backtracked to is v_3 in $\neg\rho_1$. As v_3 has no other successor locations in $\neg H$, the DFS keeps backtracking to v_5 and traverses the next path $\neg\rho_2 = \langle v_6\rangle \xrightarrow[e_5]{} \langle v_5\rangle \xrightarrow[e_8]{} \langle v_1\rangle$. Similarly, LP solver locates the infeasible core path segment of $\neg\rho_2$ as $\neg\rho_2' = \langle v_6\rangle \xrightarrow[e_5]{} \langle v_5\rangle \xrightarrow[e_8]{} \langle v_1\rangle$. In this case, the DFS backtracks to v_5 in $\neg H$ again and finds there is no more path to travel. This means there doesn't exist a feasible path which can go from the initial location to the target. Therefore, the target location v_6 can be proved to be not reachable in general, not only within bound 20!

Clearly, in this example, the DFS tree that needed to be traversed are shrunk significantly by B-DFS BMC. Many paths like $(\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle)^k \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$ in F-DFS are pruned. The B-DFS only needs to check two paths at all to tell location v_6 is unreachable.

C. Bidirectional-DFS Implementation

As discussed above, if the path segment which makes the whole path infeasible is "deep" or has many entry points, the B-DFS BMC method will be more efficient than the F-DFS BMC method. Nevertheless, in another word, if the path segment is "shallow" or has many exit points, then F-DFS BMC will outperform B-DFS BMC. So, none of these two techniques can take over the other one.

Therefore, we combine the F-DFS and B-DFS BMC to together, which results in a bidirectional-DFS BMC. In our bidirectional-DFS, the algorithm will start two threads, one conducts the F-DFS BMC on the original LHA and the other conducts the B-DFS on the reversed model. Be different from classical bidirectional-DFS algorithms like[11], [12], where the two threads can cooperate with each other, in our algorithm, these two threads work in a competition nature, that these two threads will not communicate with each other during searching, and the algorithm terminates when any of these two threads finish the searching. Under this setting, no matter where the infeasible path segment is, our bidirectional-DFS BMC can traverse the state space efficiently. The pseudo code for our implementation is shown in Table. I.

TABLE I
PSEUDO CODE FOR BIDIRECTIONAL-DFS BMC

```
SOLVER (HybridAutomata ha)
 1 .  HybridAutomata $ha_r$=reverse(ha);
 2 .  Thread $t_1$=new subSolver(ha);
 3 .  Thread $t_2$=new subSolver($ha_r$);
 4 .  while $t_1$.isAlive() and $t_2$.isAlive()
 5 .      sleep(10);
 6 .  if ($t_1$.isAlive())
 7 .      return $t_2$.result();
 8 .  else
 9 .      return $t_1$.result();
10 .  return 0;
```

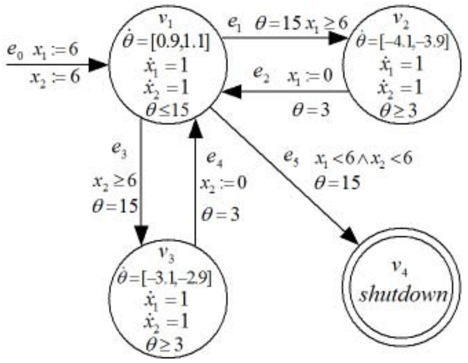
III. CASE STUDIES

We implement the bidirectional-DFS BMC, (F-DFS plus B-DFS), into our LHA bounded model checker BACH [15], [16] as guided by the last section. The latest version of BACH (V4.0) is implemented in Java, and can be downloaded from http://seg.nju.edu.cn/BACH/. As the LP solver underlying the previous versions of BACH is OR-objects[19] which does not support the functionality of IIS analysis, BACH 4 calls the IBM CPLEX[20] instead, which gives a nice support of IIS analysis. The main functionality of BACH is provided by the following set of services:

- Graphical LHA Editor: This component allows users to construct, edit, and perform syntax analysis of LHA interactively. This Editor can also transform the graphical representation of LHA to a readable text file which is used as the input file for reachability checking.
- Path-Oriented Reachability Checker: The checker requires users to select a specific path in the model. Then, it can check whether the reachability specification is satisfied along with the given path.
- Bounded Reachability Checker: This checker uses the path-oriented checker as underlying solver. It traverses the behavior tree of the model under the threshold by our adapted DFS algorithm, and checks the related path for reachability to perform bounded reachability checking.

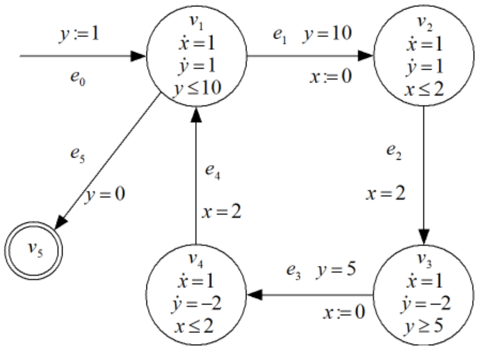
In order to evaluate the performance of the BACH 4.0, we conduct a series of case studies on a set of well-known cases, which includes the temperature control system in Fig.3 and water-level monitor system in Fig.4, the scalable automated highway system in Fig.5, and the sample automaton given in Fig.1. The target locations are all marked by double circle in the models and they are all unreachable.

We compare the BMC performance of BACH 4.0, marked as $BACH_{F+B}$, with the previous BACH, which only includes F-DFS and marked as $BACH_F$, and the-state-of-the-art SMT solver MathSAT5[9]. The experiments are conducted on a Think Center desktop machine(Intel Core2 Quad CPU 2.83GHz, 4GB RAM and Ubuntu 10.04). The time limit for experiment is set as one hour. The input models we used in experiments are all available from http://seg.nju.edu.cn/BACH/.



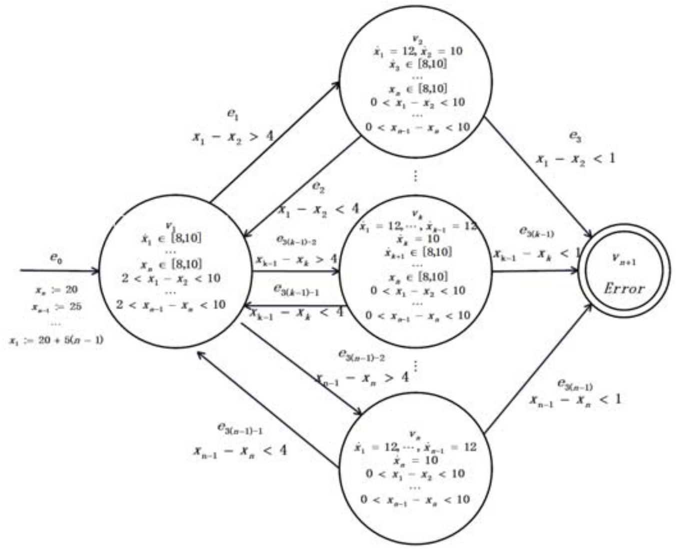
e_0 $x_1 := 6$ $x_2 := 6$
v_1 $\dot{\theta} = [0.9, 1.1]$ $\dot{x}_1 = 1$ $\dot{x}_2 = 1$ $\theta \le 15$
e_1 $\theta = 15$ $x_1 \ge 6$
v_2 $\dot{\theta} = [-4.1, -3.9]$ $\dot{x}_1 = 1$ $\dot{x}_2 = 1$ $\theta \ge 3$
e_2 $x_1 := 0$ $\theta = 3$
e_3 $x_2 \ge 6$ $\theta = 15$
e_4 $x_2 := 0$ $\theta = 3$
e_5 $x_1 < 6 \wedge x_2 < 6$ $\theta = 15$
v_3 $\dot{\theta} = [-3.1, -2.9]$ $\dot{x}_1 = 1$ $\dot{x}_2 = 1$ $\theta \ge 3$
v_4 shutdown

Fig. 3. Temperature Control System



$y := 1$
e_0
v_1 $\dot{x} = 1$ $\dot{y} = 1$ $y \le 10$
e_1 $y = 10$ $x := 0$
v_2 $\dot{x} = 1$ $\dot{y} = 1$ $x \le 2$
e_5 $y = 0$
e_4 $x = 2$
e_2 $x = 2$
v_5
v_4 $\dot{x} = 1$ $\dot{y} = -2$ $x \le 2$
e_3 $y = 5$ $x := 0$
v_3 $\dot{x} = 1$ $\dot{y} = -2$ $y \ge 5$

Fig. 4. Water-Level Monitor System



e_0
$x_n := 20$ $x_{n-1} := 25$ ... $x_1 := 20 + 5(n-1)$
v_1 $x_1 \in [8,10]$ ... $x_n \in [8,10]$ $2 < x_1 - x_2 < 10$ ... $2 < x_{n-1} - x_n < 10$
e_1 $x_1 - x_2 > 4$
v_2 $\dot{x}_1 = 12, \dot{x}_2 = 10$ $x_3 \in [8,10]$ ... $x_n \in [8,10]$ $0 < x_1 - x_2 < 10$ ... $0 < x_{n-1} - x_n < 10$
e_2 $x_1 - x_2 < 4$
e_3 $x_1 - x_2 < 1$
$e_{3(k-1)-2}$ $x_{k-1} - x_k > 4$
$e_{3(k-1)-1}$ $x_{k-1} - x_k < 4$
v_k $\dot{x}_1 = 12, \cdots, \dot{x}_{k-1} = 12$ $\dot{x}_k = 10$ $x_{k+1} \in [8,10]$ ... $x_n \in [8,10]$ $0 < x_1 - x_2 < 10$ ... $0 < x_{n-1} - x_n < 10$
$e_{3(k-1)}$ $x_{k-1} - x_k < 1$
v_{n+1} Error
$e_{3(n-1)-2}$ $x_{n-1} - x_n > 4$
$e_{3(n-1)-1}$ $x_{n-1} - x_n < 4$
$e_{3(n-1)}$ $x_{n-1} - x_n < 1$
v_n $\dot{x}_1 = 12, \cdots, \dot{x}_{n-1} = 12$ $\dot{x}_n = 10$ $0 < x_1 - x_2 < 10$ ... $0 < x_{n-1} - x_n < 10$

Fig. 5. Automated Highway System

The experiment data for the time (second) spent in each benchmark is shown below in Table.II-V respectively. If the checker fails to give a result in the time limit, the corresponding blank is marked as N/A. The number of bound means the largest number of discrete locations that a path can have in the state space under searching. Furthermore, as we mentioned that if the DFS finished before touching the bound, then BACH can prove the target is not reachable in general. In such cases, the time that BACH spent for solving the problem are marked with subscript G. For example, for the sample automaton, no matter how large the bound is, the new BACH only needs to check two paths to tell that the location v_6 is not reachable in general which only took 0.01 second. Therefore, in Table.II, all the blanks in column $BACH_{F+B}$ are combined to together. Besides of that, a special row ∞ is added to emphasise this problem can be proved in general not only in given bound.

As we can see from these tables that, for sample, water and highway system, new BACH can all give a general proof of the unreachability of the targets. Therefore, we make a comparison of new BACH with PHAVer[7] which is the-state-of-the-art classical model checker for LHA. We find that for sample automaton and water automaton, both BACH and PHAVer can finish in 0.01 second. For highway system which size, number of locations and variables, is scalable by increasing the number of vehicles in the system, the experimental data is plotted in Fig.6. We can see that the largest highway system that new BACH solved in one hour has 150 vehicles included, which is a big model of 150 variables and 151 locations, while in one hour, PHAVer can only solve a system with 6 vehicles. Furthermore, the only model that new BACH can not give a general proof is the temperature control system, while the computation of PHAVer can not terminate on this model and it fails to give any result about this model.

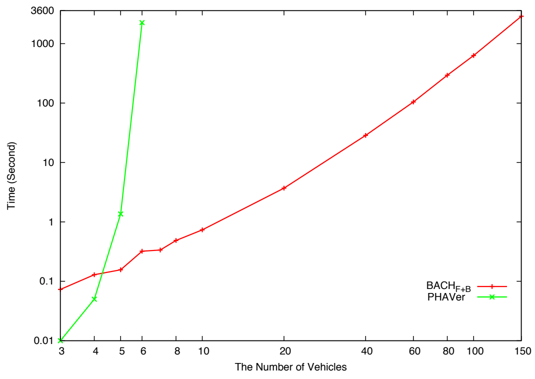


Fig. 6. BACH VS PHAVer on Automated Highway System

From these data, we can see that:

- By the help of integrating backward-DFS into BACH, the performance of BACH is optimized significantly.
- As BACH only checks one path at a time, the complexity of the verification is well controlled. As a result, the scalability of BACH is much better than the SMT-style BMC solvers, like MathSAT.
- When the DFS terminates before touching the bound, DFS-style BMC can prove the unreachability of certain

TABLE II
Performance On The Sample Automaton

Bound \ *Tech.*	$BACH_F$	$BACH_{F+B}$	MathSAT
40	0.798		4.44
60	86.303		17.977
80	N/A	0.01_G	38.006
200	N/A		2597.77
∞	N/A		N/A

TABLE III
Performance On The Tempreture Control System

Bound \ *Tech.*	$BACH_F$	$BACH_{F+B}$	MathSAT
20	0.201	0.136	0.748
40	140.618	0.665	7.896
100	N/A	6.903	613.774
600	N/A	1399.473	N/A

TABLE IV
Performance On The Water-level Monitor Automaton

Bound \ *Tech.*	$BACH_F$	$BACH_{F+B}$	MathSAT
50	0.016		3.544
150	0.046		139.141
250	0.167	0.01_G	2050.204
8000	2194.268		N/A
∞	N/A		N/A

TABLE V
Performance On The Automated Highway System with 10 Vehicles

Bound \ *Tech.*	$BACH_F$	$BACH_{F+B}$	MathSAT
10	1.882		0.648
50	N/A		21.157
100	N/A	0.733_G	146.665
150	N/A		3100.934
∞	N/A		N/A

targets in general which is incapable for SMT-style BMC checker.

- The underlying decision procedure of BACH is LP. It is well known that LP is much cheaper than polyhedral computation which is under PHAVer. As polyhedral computation is extremely sensitive to number of variables, we can see BACH outperforms PHAVer substantially when facing system with large number of variables.

IV. Conclusion and Future Work

The state-of-the-art tools for the bounded reachability analysis of LHA can only analyze systems with small dimension and bound. In the previous work, we present a DFS-style BMC method to traverse and check each abstract path in bound in the graphical structure of the LHA. Clearly, the DFS finishes more quickly, the bounded model checking can be conducted more efficiently.

In this paper, we present a backward searching technique for the DFS-style traversing strategy and combine it with the classical forward DFS in our tool BACH to accelerate the DFS traversing. The experiments show that the size of the problem that BACH can solve is increased substantially. By this forward plus backward DFS approach, BACH outperforms the-state-of-the-art competitors significantly.

In the current setting, the F-DFS and B-DFS are running independently. In the future, we will try to let these two threads to cooperate with each other. Then the state space has been pruned by one thread can benefits the other one, and the DFS-based BMC shall be finished more efficiently.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No.61100036, No.90818022, and No.61170066), the National Grand Fundamental Research 973 Program of China (No.2009CB320702), the National 863 High-Tech Programme of China (No.2011AA010103, No.2012AA011205) and by the Jiangsu Province Research Foundation (No.BK2011558).

References

[1] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of LICS 1996*, IEEE Computer Society Press, 1996, pp. 278-292.
[2] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? In *Journal of Computer and System Sciences*, 57:94-124, 1998.
[3] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H.Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. In *Theoretical Computer Science*, 138(1995), pp.3-34.
[4] E. Clarke, O. Grumburg, and D. Peled. Model Checking. The MIT Press, 1999.
[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, Y. Zhu. Bounded Model Checking. In *Advance in Computers*, Vol.58, Academic Press, 2003, pp.118-149
[6] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. In *STTT*, 1:110-122, Springer, 1997.
[7] G. Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. In *Proceedings of HSCC'05*, LNCS 2289, pp.258-273.
[8] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. In *Journal on Satisfiability, Boolean Modeling and Computation*, 2007, vol.1, pp.209-236
[9] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, Roberto Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. In *Proceedings of BMC 04*, ENTCS 119:2, Elsevier Science, 2005, pp. 17-32.
[10] Ofer Strichman. Accelerating Bounded Model Checking of Safety Properties", In *Formal Methods for System Design*, 24, pp.5-24, 2004.
[11] Dennis de Champeaux, Lenie Sint. An improved bidirectional heuristic search algorithm, In *Journal of the ACM* 24:(2), pp.177-191, 1977.
[12] Ira Pohl. Bi-directional Search, In *Machine Intelligence*, 6, Edinburgh University Press, pp.127C140, 1971.
[13] X. Li, S. Jha, and L. Bu. Towards an Efficient Path-Oriented Tool for Bounded Reachability Analysis of Linear Hybrid Systems using Linear Programming. In *Proceedings of BMC06*, ENTCS 174:3, Elsevier Science, 2007, pp.57-70.
[14] L. Bu, and X. Li. Path-Oriented Bounded Reachability Analysis of Composed Linear Hybrid Systems, In *Software Tools Technology Transfer*, 13:4, pp.307-317, Springer, 2011.
[15] L. Bu, Y. Li, L. Wang and X. Li. BACH: Bounded Reachability Checker for Linear Hybrid Automata. In *Proceedings of FMCAD 08*, IEEE Computer Society, pp.65-68,2008.
[16] L. Bu, Y. Li, L. Wang, X. Chen and X. Li. BACH 2: Bounded ReachAbility CHecker for Compositional Linear Hybrid Systems, In *Proceedings of the 13th Design Automation & Test in Europe Conference*, Dresden, Germany, pp. 1512-1517, 2010.
[17] L. Bu, Y. Yang and X. Li. IIS-Guided DFS For Efficient Bounded Reachability Analysis of Linear Hybrid Automata, In *Proceedings of HVC 2011*.
[18] Chinneck, J., Dravnieks, E. Locating minimal infeasible constraint sets in linear programs. In *ORSA Journal on Computing*, 3 (1991), 157-168.
[19] OR-Objects. http://OpsResearch.com/ OR-Objects/index.html.
[20] CPLEX. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/

Author Index