Hana Chockler
Alan J. Hu (Eds.)

LNCS 5394

Hardware and Software: Verification and Testing

4th International Haifa Verification Conference, HVC 2008
Haifa, Israel, October 2008
Proceedings

Springer

Hana Chockler   Alan J. Hu (Eds.)

Hardware and Software: Verification and Testing

4th International
Haifa Verification Conference, HVC 2008
Haifa, Israel, October 27-30, 2008
Proceedings

Springer

Volume Editors

Hana Chockler
Haifa University, IBM Haifa Labs
Haifa, 31905, Israel
E-mail: hanac@il.ibm.com

Alan J. Hu
University of British Columbia
Department of Computer Science
Vancouver BC V6T 1Z4, Canada
E-mail: ajh@cs.ubc.ca

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2.4-5, D.2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-01701-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-01701-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12655274 06/3180 5 4 3 2 1 0

Preface

These are the conference proceedings of the 4th Haifa Verification Conference, held October 27–30, 2008 in Haifa, Israel. This international conference is a unique venue that brings together leading researchers and practitioners of both formal and dynamic verification, for both hardware and software systems.

This year's conference extended the successes of the previous years, with a large jump in the number of submitted papers. We received 49 total submissions, with many more high-quality papers than we had room to accept. Submissions came from 19 different countries, reflecting the growing international visibility of the conference. Of the 49 submissions, 43 were regular papers, 2 of which were later withdrawn, and 6 were tool papers. After a rigorous review process, in which each paper received at least four independent reviews from the distinguished Program Committee, we accepted 12 regular papers and 4 tools papers for presentation at the conference and inclusion in this volume. These numbers give acceptance rates of 29% for regular papers and 67% for tool papers (34% combined) — comparable to the elite, much older, conferences in the field. A Best Paper Award, selected on the basis of the reviews and scores from the Program Committee, was presented to Edmund Clarke, Alexandre Donzé, and Axel Legay for their paper entitled "Statistical Model Checking of Mixed-Analog Circuits with an Application to a Third-Order Delta-Sigma Modulator."

The refereed program was complemented by an outstanding program of invited talks, panels, and special sessions from prominent leaders in the field. We have included in this volume abstracts, and papers if available, from the invited program. Among the invited program was the recipient of the 2008 HVC Award, chosen as the most influential work in the past five years in the field of verification. This year's winner was Ken McMillan, for his work on interpolants.

A conference of this scope happens only through the tireless contributions of many people. On the technical side, we are grateful to the Program Committee and their many additional reviewers for ensuring the intellectual quality of the conference. We thank the HVC Sesssion Chairs (in the order of appearance): Ken McMillan, Malay Ganai, Moshe Vardi, Daniel Jackson, Carl Pixley, Doron Peled, Jason Baumgartner, and Karen Yorav, for their knowledgeable and professional chairing of the sessions. We are especially thankful to organizers and chairs of the special sessions: Orna Grümberg, who organized and chaired a panel on coverage "across the verification domain," and to Ziyad Hanna and Warren Hunt, who organized and chaired a special session on post-silicon verification. We also thank the HVC Award Committee, who tackled the unenviable task of selecting a single winner from several extraordinary works. On the logistical side, special thanks go to Vered Aharon for mastery of the countless organizational issues that needed to be addressed. We also thank IBM for providing administrative support and services such as graphic design, technical writing, printing, and, of course, a cafeteria, free of charge to the conference participants. We would also like to thank the HVC Organizing Committee, who are an endless source of knowledge, wisdom, and guidance. Finally, thank you to everyone who participated in the conference: a successful conference is a unique little neighborhood in space and time, and it is the participants who create the magic of the moment.

October 2008

Alan J. Hu
Hana Chockler

Organization

Conference Chairs

Hana Chockler	IBM Haifa Research Lab, Israel	General Chair
Alan J. Hu	University of British Columbia, Canada	Program Chair

Organizing Committee

Sharon Barner	IBM Haifa Research Lab, Israel
David Bernstein	IBM Haifa Research Lab, Israel
Laurent Fournier	IBM Haifa Research Lab, Israel
Moshe Levinger	IBM Haifa Research Lab, Israel
Shmuel Ur	IBM Haifa Research Lab, Israel
Avi Ziv	IBM Haifa Research Lab, Israel

Program Committee

Sharon Barner	IBM Haifa Research Lab, Israel
Eyal Bin	IBM Haifa Research Lab, Israel
Roderick Bloem	Graz University of Technology, Austria
Michael Browne	IBM Poughkeepsie, USA
Hana Chockler	IBM Haifa Research Lab, Israel
Jong-Deok Choi	Samsung Electronics, Korea
Alessandro Cimatti	IRST, Italy
Kerstin Eder	University of Bristol, UK
E. Allen Emerson	University of Texas at Austin, USA
Bernd Finkbeiner	Universität des Saarlandes, Germany
Limor Fix	Intel, USA
Laurent Fournier	IBM Haifa Research Lab, Israel
Steven M. German	IBM Watson, USA
Orna Grumberg	Technion, Israel
Aarti Gupta	NEC Labs America, USA
Ziyad Hanna	Jasper Design Automation, USA
Klaus Havelund	NASA's Jet Propulsion Laboratory, Caltech
Alan Hu	University of British Columbia, Canada
Warren Hunt	University of Texas, Austin, USA
Daniel Kroening	ETH Zürich, Switzerland
Tsvi Kuflik	University of Haifa, Israel
Orna Kupferman	Hebrew University, Israel
Mark Last	Ben-Gurion University of the Negev, Israel
João Lourenço	Universidade Nova de Lisboa, Portugal
Sharad Malik	Princeton University, USA

Erich Marschner	Cadence, USA
Ken McMillan	Cadence, USA
Amos Noy	Cadence, USA
Amit Paradkar	IBM Watson, USA
Viresh Paruthi	IBM, USA
Carl Pixley	Synopsys, USA
Andrew Piziali	USA
Wolfgang Roesner	IBM Austin, USA
Fabio Somenzi	University of Colorado, USA
Scott D. Stoller	Stony Brook University, USA
Ofer Strichman	Technion, Israel
Serdar Tasiran	Koc University, Turkey
Shmuel Ur	IBM Haifa Research Lab, Israel
Willem Visser	SEVEN Networks, USA
Tao Xie	North Carolina State University, USA
Karen Yorav	IBM Haifa Research Lab, Israel

HVC Award Committee

Corina Pasareanu	Perot Systems/NASA Ames Research Center, USA (Chair)
Roderick Bloem	Graz University of Technology, Austria
Sebastian Elbaum	University of Nebraska, Lincoln, USA
Bob Kurshan	Cadence, USA
Wolfram Schulte	Microsoft Research, Redmond, USA
Willem Visser	SEVEN Networks, USA
Avi Ziv	IBM Haifa Research Lab, Israel

Additional Reviewers

Allon Adir
Nina Amla
Cyrille Artho
Jason Baumgartner
Jesse Bingham
Nicolas Blanc
Rachel Brill
Angelo Brillout
Doron Bustan
Michael L. Case
Yury Chebiryak
Vijay D'Silva
Jared Davis
Rayna Dimitrova
Klaus Dräger
Ruediger Ehlers
Cindy Eisner
Tayfun Elmas
Dana Fisman
Malay Ganai
Alberto Griggio
Klaus Havelund
Alexander Ivrii
Geert Janssen
Robert L. Kanzelman

Sponsors

The Organizing Committee of HVC 2008 gratefully acknowledges the generous financial support of:

– IBM Haifa Research Lab
– Cadence Israel
– Mentor Graphics
– Synopsys

Table of Contents

Hazards of Verification

Daniel Jackson

MIT CSAIL

It is insufficiently considered that men more often require to be reminded than informed.

Samuel Johnson

Abstract. Great progress has been made in software verification. One has only to look, for example, at the growth in power of SAT and its widening class of applications to realize that concerns early on that verification was fundamentally intractable and that it would remain a largely manual activity were mistaken. Nevertheless, despite many research advances, verification is still not widely applied to software. In order to make verification practical, some fundamental obstacles will need to be addressed. This talk outlined a sample of these obstacles, in three categories: technical obstacles in the mechanism of verification itself; engineering obstacles related to the problem of establishing confidence in a system rather than a single component; and social and managerial obstacles related to the process of verification.

These obstacles are known and have been fairly widely discussed. It is useful, however, to remind ourselves of them since otherwise, as a research community, we risk focusing on technical details that may be overwhelmed by these larger factors. In explaining some of the obstacles, I also sought to question some assumptions that are often made in the verification community that make sense in a narrower, mathematical context but are not defensible in the larger system context.

One example of this is the notion of conservatism. It is often assumed that a 'conservative' analysis that never misses bugs, but which may generate false alarms, is necessarily superior to an analysis that is 'unsound' and may fail to report real errors. In practice, however, this superiority is not inevitable, and in many cases, an unsound analysis is preferable. First, the need to be conservative can result in a dramatic increase in false positives, so that the unsound analysis may actually be more accurate (in the sense that the probability of missing a bug is low, whereas the comparable conservative analysis produces a report full of erroneous claims). Second, the presence of false alarms may increase the burden of reading the report so dramatically that bugs may be overlooked in the filtering process, so that the conservative analysis becomes, in its context of use, unsound. Similarly, in a software design context, the assumption that checks should be conservative is often not justified in engineering terms. In air-traffic control, for example, it has long been recognized that a conflict detection tool must not generate false alarms. The catastrophic accident in Guam in 1997, in which a 747 flew into the ground, might not have occurred had a safe-altitude-warning system not been disabled because of its over-conservative error reporting.

Automata-Theoretic Model Checking Revisited

Moshe Y. Vardi

Rice University, Department of Computer Science,
Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu

Abstract. In automata-theoretic model checking we compose the design under verification with a Büchi automaton that accepts traces violating the specification. We then use graph algorithms to search for a counterexample trace. The theory of this approach originated in the 1980s, and the basic algorithms were developed during the 1990s. Both explicit and symbolic implementations, such as SPIN and SMV, are widely used.

Proofs, Interpolants, and Relevance Heuristics
(HVC 2008 Award Winner)

Ken McMillan

Cadence Berkeley Labs

Abstract. The technique of Craig interpolation provides a means of extracting from a proof about a bounded execution of a system the necessary information to construct a proof about unbounded executions. This allows us to exploit the relevance heuristics that are built into modern SAT solvers and SAT-based decision procedures to scale model checking to larger systems.

This talk will cover applications of Craig interpolation in various domains, including hardware verification using propositional logic, and software verification using first-order logic.

Is Verification Getting Too Complex?

Yoav Hollander

Cadence Israel

Abstract. Verification of HW and HW/SW systems is becoming more and more complex. This presentation will look at the reasons for the increased complexity and what can be done about it.

Specifically, I shall look at where bugs come from and how they flow through the system (from specification, through design, detection, debug and removal). I shall then present some promising directions for reducing verification complexity, many of them attacking “incidental complexity”.

Throughout the presentation, I shall try to emphasize currently-neglected research areas.

Can Mutation Analysis Help Fix Our Broken Coverage Metrics?

Brian Bailey

Abstract. The semiconductor industry relies on coverage metrics as its primary means of gauging both quality and readiness of a chip for production, and yet the metrics in use today measure neither quality nor provide an objective measure of completeness. This talk will explore the problems with existing metrics and why they are not proper measures of verification. Mutation analysis looks like a promising technology to help bridge the divide between what we have and what we need in terms of metrics and may also be able to help bridge the divide between static and dynamic verification. The talk will conclude with some of the remaining challenges that have to be overcome, such as its correct fit within a verification methodology and the standardization of a fault model.

Practical Considerations Concerning HL-to -RT Equivalence Checking

Carl Pixley

Synopsys

Abstract. We will discuss several years' experience with commercial HL-to-RTL equivalence checking with the Hector technology. We will also discuss several considerations based upon the reality that our company is an EDA vendor. This is quite different from the position of a semiconductor company, which can concentrate on a very specific methodology and design type.

Our observations will include some case studies from customers about the methodology and designs on which they are using Hector. Most of the development of Hector was based upon solutions to problems presented by our customers. We will also discuss the general architecture of Hector and some technological information about the engines that underlie Hector.

A Framework for Inherent Vacuity

Dana Fisman[1,2,⋆], Orna Kupferman[1], Sarai Sheinvald-Faragy[1], and Moshe Y. Vardi[3,⋆⋆]

[1] School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel
[2] IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
[3] Rice University, Houston Texas 77005, USA

Abstract. Vacuity checking is traditionally performed after model checking has terminated successfully. It ensures that all the elements of the specification have played a role in its satisfaction by the design. Vacuity checking gets as input both design and specification, and is based on an in-depth investigation of the relation between them. Vacuity checking has been proven to be very useful in detecting errors in the modeling of the design or the specification. The need to check the quality of specifications is even more acute in *property-based design*, where the specification is the only input, serving as a basis to the development of the system. Current work on property assurance suggests various sanity checks, mostly based on satisfiability, non-validity, and realizability, but lacks a general framework for reasoning about the quality of specifications.

We describe a framework for *inherent vacuity*, which carries the theory of vacuity in model checking to the setting of property-based design. Essentially, a specification is inherently vacuous if it can be mutated into a simpler equivalent specification, which we show to coincide with the fact the specification is satisfied vacuously in all systems. We also study the complexity of detecting inherent vacuity, and conclude that while inherent vacuity leads to specifications that better capture designer intent, it is not more complex than simple property-assurance checks.

1 Introduction

In recent years, we see a growing awareness to the importance of assessing the quality of (formal) specifications. In the context of model checking, a specification consists of a set of formulas written in some temporal logic, and the quality of the specification is assessed by analyzing the effect of applying mutations to the formulas. If the system satisfies the mutated specification, we know that some elements of the specification do not play a role in its satisfaction, thus the specification is satisfied in some *vacuous* way [3, 20]. Vacuity is successfully used in order to improve specifications and detect design errors [18].

The need to assess the quality of specifications is even more acute in the context of *property-based design* [24]. There, the design process starts with the development

⋆ The work of this author was done as part of the Valazzi-Pikovsky Fellowship Fund.
⋆⋆ Supported in part by NSF grants CCR-0124077, CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by gift from Intel.

of the specification as a set of temporal formulas, which then serves as a basis to the development of the implementation. For example, in *temporal synthesis* [23], we go automatically from the specification to a system that satisfies it. Indeed, one of the criticisms against synthesis is that it does not eliminate the difficulty of design, but merely shifts the difficulty of developing correct implementations to that of developing correct specifications [17].

Property assurance is the activity of eliciting specifications that faithfully capture designer intent [5, 27]. Obvious quality checks one may perform for a given specification are *non-validity* and *satisfiability* [28]. More involved quality checks are studied in the PROSYD project [24]. There, one considers a set of temporal formulas, partitioned into assumptions and guarantees, and checks consistency of the assumptions and various types of entailment of guarantees by assumptions. Recent work has focused on other aspects of property assurance. For example, both [10, 22] study *completeness analysis for property sets*. There, one analyzes a specification consisting of a set of temporal formulas and measures the degree to which the specification determines the exact behavior of each of its signals. There is a trade-off between the level of abstraction that a specification enjoys and the level of detail in which it describes the system. The analysis in [10, 22] takes one side of this trade-off, as it expects the specifications to determine the exact behavior of the signals.[1]

As discussed in [27], checking vacuity of the formulas in the context of property assurance would be of great importance. While vacuity has been widely studied in the context of model checking [2, 3, 6, 14, 20, 21], it is not clear how to define and check vacuity of formulas without having a system that is meant to satisfy these formulas. A first step for analyzing vacuity in a specification is taken in [9], which studies *early detection of vacuity* and provides the inspiration to this work. There too, a specification consists of a set of temporal formulas, and the goal is to reduce vacuity of the specification by removing formulas that are implied by the specification, and by strengthening formulas to ones that are still implied by the specification. While [9] introduced the intuitive concept of "vacuity without design", it does not attempt to define this concept. Rather, it offers various sanity checks that can be applied to sets of properties, with the aim of simplifying later vacuity checking with respect to a design. Our aim in this work is to formalize the intuitive concept introduced in [9].

We describe a framework for *inherent vacuity* for sets of linear temporal properties. The term "inherent" refers to the fact that we do not study vacuity of properties with respect to a given system, but as a quality measure of the properties themselves. We focus on both identifying the appropriate definition of inherent vacuity, as well as developing algorithms for testing inherent vacuity.

Before we present our definition for inherent vacuity, let us recall one definition of vacuity in LTL model checking [2]. There, given a system $\mathcal{S}$ and a specification φ that

[1] A related line of research is that of *specification debugging* [1], where, in the process of model checking, counterexamples are automatically clustered together in order to make the manual debugging of temporal properties easier. Another related line of research is that of *coverage metrics* [8, 16]. There, the mutations are applied to the system, and if the mutated system satisfies the specification, we know that some elements of the system are not covered by the specification.

is satisfied in $\mathcal{S}$, we say that a subformula ψ of φ *does not affect* the satisfaction of φ in $\mathcal{S}$ if $\mathcal{S}$ also satisfies the stronger specification $\forall x.\varphi[\psi \leftarrow x]$, in which ψ is replaced by a universally quantified proposition. Intuitively, this means that $\mathcal{S}$ satisfies φ even with the most challenging assignments to ψ.[2] The specifications φ is then *vacuously satisfied* in $\mathcal{S}$ if it has a subformula that does not affect its satisfaction in $\mathcal{S}$.

There are two natural approaches to lift the definition of vacuity in the context of model checking to a definition of inherent vacuity. In order to see the idea behind the first approach, consider specifications that are tautologies or contradictions. One need not have a context in order to see that they fail any reasonable criterion, and indeed non-validity and satisfiability checking are useful sanity checks [28]. The validity criterion is a special case of a weaker criterion, in which a specification fails if we can mutate it and get a simpler, equivalent specification. For tautologies, the mutation yields the specification *true*. Our criteria use less aggressive mutations, and are inspired by the concept of vacuity in model checking. We say that a specification φ is inherently vacuous if φ is equivalent to $\forall x.\varphi[\psi \leftarrow x]$, for some subformula ψ of φ. For example, the specification $\mathsf{G}\,(busy \rightarrow \mathsf{F}\,grant) \wedge \mathsf{G}\,(\neg busy \rightarrow \mathsf{F}\,grant)$ is inherently vacuous, as it is equivalent to the specification $\forall x.\mathsf{G}\,(x \rightarrow \mathsf{F}\,grant) \wedge \mathsf{G}\,(\neg x \rightarrow \mathsf{F}\,grant)$, which is equivalent to $\mathsf{G}\,\mathsf{F}\,grant$. This approach leads to a PSPACE decision procedure for inherent vacuity for LTL specifications, by reducing it to the satisfiability problem for LTL.

As described above, our first approach for defining inherent vacuity is based on the definition of vacuity in model checking, and it adopts the idea of applying mutations to the specification. With no system to check the mutated specification with respect to, our first approach requires the mutated specification to be equivalent to the original one. Our second approach for defining inherent vacuity, also based on the definition of vacuity in model checking, quantifies the missing context (that is, the system) universally. Thus, according to the second approach, a specification φ is inherently vacuous if φ is satisfied vacuously in all systems that satisfy it. Note that in contrast to the first definition, the definition does not require the same subformula not to affect the satisfaction of the specification in all systems. Keeping in mind the trade-off between abstraction and vacuity, one may welcome specifications that are vacuously satisfied according to the second approach yet have no single subformula that does not affect the satisfaction in all systems. We show, however, that the second approach coincides with the first one. Thus, a specification φ is satisfied vacuously in all systems that satisfy it iff φ is equivalent to some mutation of it.

The above two approaches, and the encouraging fact they coincide, set the base to our framework for inherent vacuity. Experience with vacuity in model checking has led to the conclusion that there is no single definition of vacuity that is superior to all others, and various definitions are used in practice [2–4, 6, 7, 11, 15, 20, 21, 31]. Our framework refines the definition of inherent vacuity to account not only for the different definitions of vacuity in model checking, but also for the different settings in which property-based design is used (closed vs. open systems), the goal of the designer (tightening the specification or only cleaning it), and the polarity of the vacuity (strengthening vs. weakening of the formula). Thus, we do not offer a single definition of inherent vacuity, but, rather,

[2] Since ψ may have several occurrences in φ there need not be a single "most challenging assignment" and it need not be *true* or *false*.

offer a general framework in which the user can choose the parameters that best suit the application. We view the main practical contribution of the paper in the setting of open systems and temporal synthesis. As discussed above, the problem of eliciting specifications that faithfully capture the designer intent is of great importance in this setting. Still, the only sanity check that is now used for a specification of an open system is its *realizability*, which is analogous to satisfiability in the setting of closed systems, and checks that there is at least one open system that satisfies the specification [23].

We show that in all variations, the two approaches to defining inherent vacuity coincide. We study the problem of deciding whether an LTL formula is inherently vacuous according to the various criteria and settings. We show that the problem is related to the basic problem in the corresponding setting. Thus, for the setting of closed system, the problem can be solved in PSPACE, just like the satisfiability problem [29], while for the setting of open systems, the problem can be solved in 2EXPTIME, just like the realizability problem [26]. Thus, detection of inherent vacuity is not harder than the most basic quality checks for specifications. We provide many examples for inherent vacuity and argue for the likelihood of encountering inherent vacuity in real life specifications.

2 Inherent Vacuity

In this section we define inherent vacuity for LTL formulas and study basic properties of the definition.

We first review the definition of LTL vacuity in model checking. We assume the reader is familiar with the syntax and the semantics of LTL. The semantic approach to LTL vacuity in model checking [2] considers LTL formulas augmented with universal quantification over atomic propositions. Recall that an LTL formula over a set AP of atomic propositions is interpreted over computations of the form $\pi = \pi_0, \pi_1, \pi_2, \ldots$, with $\pi_i \subseteq AP$. The computation then satisfies a formula of the form $\forall x.\varphi$, where φ is an LTL formula and x is an atomic proposition, if φ is satisfied in all the computations that agree with π on all the atomic propositions except (maybe) x. Thus, $\pi \models \forall x.\varphi$ iff $\pi' \models \varphi$ for all $\pi' = \pi'_0, \pi'_1, \pi'_2, \ldots$ such that $\pi'_i \cap (AP \setminus \{x\}) = \pi_i \cap (AP \setminus \{x\})$ for all $i \geq 0$. As with LTL, a Kripke structure $\mathcal{K}$ satisfies $\forall x.\varphi$ if all computations of $\mathcal{K}$ satisfy $\forall x.\varphi$. For two LTL formulas φ and φ', we say that φ and φ' are *equivalent*, denoted $\varphi \equiv \varphi'$, if for every Kripke structure $\mathcal{K}$, we have that $\mathcal{K} \models \varphi$ iff $\mathcal{K} \models \varphi'$.

Given a Kripke structure $\mathcal{K}$ and a formula φ satisfied in $\mathcal{K}$, we say that a subformula ψ of φ *does not affect the satisfaction of* φ *in* $\mathcal{K}$ if $\mathcal{K}$ also satisfies the formula $\forall x.\varphi[\psi \leftarrow x]$, in which ψ is replaced by a universally quantified fresh proposition [2]. Intuitively, this means that $\mathcal{K}$ satisfies φ even with the most challenging assignments to ψ. We refer to the formula $\forall x.\varphi[\psi \leftarrow x]$ as the *ψ-strengthening* of φ. Finally, a formula φ is *vacuously satisfied in* $\mathcal{K}$ if φ has a subformula that does not affect its satisfaction in $\mathcal{K}$.

In the context of inherent vacuity, the Kripke structure $\mathcal{K}$ is not given. We are only given the formula φ, and we seek some quality criteria that would indicate the likelihood of φ to be satisfied vacuously. Below we describe two natural approaches to defining inherent vacuity, and show that they are, in fact, equivalent.

The first approach to defining inherent vacuity is based on mutating the formula. The idea is that if a syntactic manipulation on the formula, which typically changes the semantics of the formula, yields an equivalent formula, then something is inherently vacuous in the given formula.

Definition 1. *We say that an* LTL *formula φ is* inherently vacuous by mutation *if there exists a subformula ψ of φ such that $\varphi \equiv \forall x.\varphi[\psi \leftarrow x]$. That is, φ is equivalent to its ψ-strengthening. We then say that φ is inherently vacuous by mutation* with witness ψ.

Example 1. The formula $\varphi = \mathsf{F}\,(grant \vee fail) \vee \mathsf{X}\,fail$ is inherently vacuous by mutation, as it is equivalent to its $\mathsf{X}\,fail$-strengthening $\forall x.\mathsf{F}\,(grant \vee fail) \vee x$. To see this, note that both φ and its $\mathsf{X}\,fail$-strengthening are equivalent to $\mathsf{F}\,(grant \vee fail)$.

The formula $\varphi = (\neg busy \wedge (busy\,\mathsf{U}\,ack)) \vee (busy \wedge ack)$ is inherently vacuous by mutation, as it is equivalent to its $busy$-strengthening $\forall x.\varphi[busy \leftarrow x]$. To see this, note that both φ and its $busy$-strengthening are equivalent to the formula ack.

Remark 1. The purpose of our examples is to show patterns for inherently vacuous specs. Thus, while we do not expect designers to write the specifications in the examples, where the vacuity is obvious, such patterns do appear in real life specifications. Indeed, there, the specifications are more involved, and it is hard to keep track of all relations among the propositions induced by a specification. We discuss this issue in detail in Section 4.

As we show later in Theorem 2, defining inherent vacuity by means of mutations enables us to reduce the problem of deciding whether a given specification is inherently vacuous to the satisfiability problem for LTL.

It is not hard to see that Definition 1 is equivalent to a definition in which a formula is inherently vacuous if there exists a subformula ψ of φ such that ψ does not affect the satisfaction of φ in all Kripke structures that satisfy it. Formally, for every system $\mathcal{K}$ such that $\mathcal{K} \models \varphi$, also $\mathcal{K} \models \forall x.\varphi[\psi \leftarrow x]$.

One may find Definition 1 too restrictive, as it focuses on a single subformula of the specification. The second approach to defining inherent vacuity addresses this point, by starting with the definition of vacuity and quantifying the missing context (that is, the system) universally, without restricting attention to a single subformula. Formally, we have the following.

Definition 2. *We say that an* LTL *formula φ is* inherently vacuous by model *if for every Kripke structure $\mathcal{K}$, if $\mathcal{K} \models \varphi$, then $\mathcal{K}$ satisfies φ vacuously.*

There is a trade-off between the level of abstraction that a specification enjoys and the level of detail in which it describes the system. Thus, one may tolerate formulas that are vacuously satisfied yet have no single subformula to blame, and find this second approach too unrestricted. As we show now, however, in the setting of nondeterministic Kripke structures, the two approaches we defined coincide. Formally, we have the following.

Theorem 1. *An* LTL *specification φ is inherently vacuous by mutation iff φ is inherently vacuous by model.*

Proof. For the first direction, assume that φ is inherently vacuous by mutation. Then there is a subformula ψ of φ such that $\varphi \equiv \forall x.\varphi[\psi \leftarrow x]$. Accordingly, for every

Kripke structure $\mathcal{K}$, if $\mathcal{K} \models \varphi$, then $\mathcal{K} \models \forall x.\varphi[\psi \leftarrow x]$, and so $\mathcal{K}$ satisfies φ vacuously. Thus, φ is inherently vacuous by model.

For the second direction, assume that φ is inherently vacuous by model, and assume by way of contradiction that φ is not inherently vacuous by mutation. Then there exists no single subformula ψ of φ such that $\varphi \equiv \forall x.\varphi[\psi \leftarrow x]$. Consider the alternative definition to vacuity by mutation. Then there is no single subformula ψ of φ such that ψ does not affect the satisfaction of φ in every Kripke structure $\mathcal{K}$ that satisfies it.

Let k be the number of subformulas that φ has. By the assumption, for every candidate subformula ψ_i of φ, with $1 \leq i \leq k$, there is a Kripke structure $\mathcal{K}_i$ that satisfies φ (and hence, as φ is inherently vacuous by model, satisfies φ vacuously), but $\mathcal{K}_i \not\models \forall x.\varphi[\psi_i \leftarrow x]$. Let $\mathcal{K} = \mathcal{K}_1 \cup \mathcal{K}_2 \cup \cdots \cup \mathcal{K}_k$ be the disjoint union of $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$. Note that the set of initial states of $\mathcal{K}$ is the union of the sets of initial states in all structures. By the semantics of LTL, the Kripke structure $\mathcal{K}$ satisfies φ. Since φ is inherently vacuous by model, $\mathcal{K}$ satisfies φ vacuously. Let ψ_i be a subformula that does not affect φ in $\mathcal{K}$. By the semantics of LTL, the subformula ψ_i does not affect φ also in $\mathcal{K}_i$, and we reached a contradiction. □

Remark 2. A deterministic Kripke structure is a Kripke structure with a single initial state in which each state has exactly one successor. If we restrict attention to deterministic Kripke structures, then Definitions 1 and 2 do not coincide. That is, there exist formulas that are inherently vacuous by model but are not inherently vacuous by mutation. For example, consider the formula $\varphi = p \vee q$. Every deterministic Kripke structure that satisfies φ has its (single) initial state labeled either by p or by q or by both, and thus it satisfies φ vacuously. On the other hand, φ is not equivalent to any strengthening of it. Since deterministic Kripke structures are not an interesting model for a system (as they induce a single computation rather than a set of computations), we continue with the nondeterministic setting.

So, the definitions that follow from the two different approaches coincide, and we use the term *inherent vacuity* to refer to either of them. We now study the complexity of detection of inherent vacuity.

Theorem 2. *Given an* LTL *formula φ and a subformula ψ of φ, deciding whether φ is inherently vacuous with witness ψ is PSPACE-complete.*

Proof. We start with the upper bound. Consider an LTL formula φ. For every subformula ψ, it holds that $\forall x.\varphi[\psi \leftarrow x]$ implies φ. Therefore, checking whether φ is inherently vacuous with witness ψ amounts to checking whether φ implies $\forall x.\varphi[\psi \leftarrow x]$. This is done by checking the satisfiability of $\varphi \wedge \exists x.\neg\varphi[\psi \leftarrow x]$, which is satisfiable iff $\varphi \wedge \neg\varphi[\psi \leftarrow x]$ is satisfiable. The latter is an LTL formula, whose satisfiability can be checked in PSPACE. For the lower bound, it is easy to see that φ is inherently vacuous with witness φ iff $\varphi \equiv$ *false*, thus PSPACE-hardness follows from the PSPACE-hardness of LTL satisfiability. □

Now, since a formula φ is inherently vacuous iff it is inherently vacuous with witness ψ for some subformula ψ of φ, the upper bound in Theorem 2 implies the following.

Corollary 1. *The problem of deciding whether an* LTL *formula is inherently vacuous can be solved in polynomial space.*

We note that the lower bound for the problem of deciding inherent vacuity is open. The difficulties in proving a PSPACE lower bound are similar to the ones encountered in studying the complexity of vacuity detection in model checking (a PSPACE upper bound is known, yet a lower bound is open [2]).

3 A Framework for Inherent Vacuity

In Section 2 we defined inherent vacuity. Experience with vacuity in model checking has led to the conclusion that there is no single definition of vacuity that is superior to all others, and various definitions are used in practice. In this section we refine the definition of inherent vacuity to account not only for the different definitions of vacuity in model checking, but also for the different settings in which property-based design is used. The refinement is based on adding parameters that refer to the type of vacuity, the context in which the specification is used, the goal of the designer, and the polarity of the mutation. It turns out that the equivalence of the two approaches to the definition of inherent vacuity is maintained in all settings. Thus, our lifting of vacuity in model checking to inherent vacuity is robust, in the sense that it works for the many contexts in which vacuity may be checked. We elaborate on the parameters below, and we first describe them for the approach that defines inherent vacuity by mutation. As in Section 2, this approach is the basis to decision procedures for inherent vacuity.

3.1 The Parameters of the Framework

Vacuity Type. Recall that a formula φ is inherently vacuous by mutation if φ is equivalent to a ψ-strengthening of it, for some subformula ψ of φ. The definition of ψ-strengthening is induced from work on vacuity in model checking. Recall that several definitions of vacuity in model checking are studied in the literature: Definitions treating each occurrence of a subformula separately [3, 20] and their extensions (to the modal μ-calculus [11] and to PSL/SVA [6]), definitions treating all occurrences of the same subformula together [2, 14], definitions considering various distinct subformulas [15], definitions considering model checker proofs [21], definitions focusing on a certain type of reasons for vacuity (antecedent [4] and environment [7]), a definition considering vacuity grounds [30, 31], and more. In Section 2, we followed the definition of vacuity in [2]. The first parameter in our framework enables the consideration of other definitions. For example, the semantic-based definition in [2] can be refined according to different semantics of universal quantification of atomic propositions (structure vs. tree; for details see [2]). As another example, in the syntactic-based definition to vacuity in [3, 20], one mutates a single occurrence of a subformula, rather than all occurrences. Formally, given an occurrence σ of a subformula of the formula φ, the σ-strengthening of φ is the formula $\varphi[\sigma \leftarrow \bot]$, obtained by replacing the occurrence σ by *false* if σ is under an even number of negation and by *true* if σ is under an odd number of negation. Other definitions allow mutations of a subset of the occurrences, a subset of subformulas, or a subset of the atomic propositions [15].

Example 2. Consider the formula $\varphi = \mathit{grant} \vee (\mathit{up}\,\mathsf{U}\,\mathit{grant})$. Clearly, φ is equivalent to $\mathit{up}\,\mathsf{U}\,\mathit{grant}$. Indeed, φ is equivalent to $\varphi[\sigma \leftarrow \bot]$ for σ being the first occurrence of the subformula *grant*. Therefore, the σ-strengthening of φ is equivalent to φ. Note that if we had considered the semantic-based definition as we did in the previous section, the formula would not have been declared inherently vacuous, since it is not equivalent to $\forall x.\varphi[\psi \leftarrow x]$ for any subformula ψ of φ.

Consider the formula $\varphi = (\neg \mathit{busy} \wedge \mathit{ack}) \vee (\mathit{busy} \wedge \mathit{ack})$. It is easy to see that φ is equivalent to $\forall x.\varphi[\mathit{busy} \leftarrow x]$. Thus, f is inherently vacuous when considering the semantic-based definition for vacuity. On the other hand, φ is not equivalent to $\varphi[\psi \leftarrow \bot]$ for any occurrence of a subformula ψ of φ. Thus, φ is not inherently vacuous when considering the syntactic-based definition of vacuity.

Equivalence Type. Again recall that a formula φ is inherently vacuous by mutation if φ is equivalent to a ψ-strengthening of it, for some subformula ψ of φ. The definition of equivalence has to do with the context in which φ is to be used. In the context of closed systems, the semantics of φ is defined with respect to Kripke structures, thus $\varphi \equiv \varphi'$ if for all Kripke structures $\mathcal{K}$, we have that $\mathcal{K} \models \varphi$ iff $\mathcal{K} \models \varphi'$. In the context of open systems, the semantics of φ is defined according to *transducers*, and the atomic propositions in φ are partitioned into input and output signals. Before we turn to show that this requires a different notion of equivalence, let us define transducers formally.

A transducer is a tuple $\mathcal{T} = \langle \mathcal{I}, \mathcal{O}, S, \eta_0, \eta, L \rangle$, where $\mathcal{I}$ is a set of input signals, $\mathcal{O}$ is a set of output signals, S is a set of states, $\eta_0 : 2^I \to 2^S \setminus \emptyset$ is an initial transition function, $\eta : S \times 2^{\mathcal{I}} \to 2^S \setminus \emptyset$ is a transition function, and $L : S \to 2^{\mathcal{O}}$ is a labeling function. Note that $\mathcal{T}$ is responsive, in the sense that η_0 and η provide at least one initial state and successor state, respectively, for each input letter. A *run* of $\mathcal{T}$ on an input sequence $i_0 \cdot i_1 \cdot i_2 \cdots \in (2^{\mathcal{I}})^\omega$ is a sequence $s_0, s_1, s_2, \ldots$ of states such that $s_0 \in \eta_0(i_0)$ and $s_{j+1} \in \eta(s_j, i_{j+1})$ for all $j \geq 0$. A computation $w \in (2^{\mathcal{I} \cup \mathcal{O}})^\omega$ is *generated* by $\mathcal{T}$ if $w = (i_0 \cup o_0), (i_1 \cup o_1), (i_2 \cup o_2) \ldots$ is such that there is a run $s_0, s_1, s_2, \ldots$ of $\mathcal{T}$ on $i_0 \cdot i_1 \cdot i_2 \cdots$ for which $o_j = L(s_j)$ for all $j \geq 0$. Note that we consider nondeterministic transducers. A transducer $\mathcal{T}$ *realizes* an LTL formula φ, denoted $\mathcal{T} \models \varphi$, if all computations of $\mathcal{T}$ satisfy φ. We say that an LTL formula φ is *realizable* if there is a transducer that realizes φ. The synthesis problem is to construct, given an LTL formula φ, a transducer that realizes φ.

Note that a nondeterministic transducer may have several runs on an input sequence. In a *deterministic* transducer, for all $i \in 2^I$ and $s \in S$, we have $|\eta_0(i)| = 1$ and $|\eta_0(s, i)| = 1$. Thus, a deterministic transducer has a single run on each input sequence. Unlike Kripke structures, which can be viewed as transducers with $I = \emptyset$, here the deterministic model is of interest, and it induces a set of computations – one for each input sequence.

As discussed in [13], equivalence with respect to transducers is weaker than equivalence with respect to Kripke structures. Formally, given two LTL formulas φ and φ' both over signals I and O, we say that φ and φ' are *equivalent in the context of open systems* (o-equivalent, for short), denoted $\varphi \equiv_o \varphi'$, if for every transducer $\mathcal{T}$ with input I and output O, we have that $\mathcal{T} \models \varphi$ iff $\mathcal{T} \models \varphi'$. For the sake of uniformity, we now use $\equiv_c$ to denote equivalence in the context of closed systems (c-equivalence, for short; what used to be $\equiv$ in Section 2). It is not hard to see that for every two LTL formulas

φ and φ', we have that $\varphi \equiv_c \varphi'$ implies $\varphi \equiv_o \varphi'$. By [13], implication on the other direction does not hold. For example, a specification φ that restricts the input in some satisfiable way is unrealizable, and hence $\varphi \equiv_o \mathit{false}$, yet $\varphi \not\equiv_c \mathit{false}$ as φ is satisfiable.

Example 3. Consider the formula $\varphi = [\mathsf{G}\,(\mathit{busy} \rightarrow \mathsf{F}\,(\mathit{grant} \wedge \neg \mathit{busy}))] \vee \mathsf{G}\,\mathsf{F}\,\mathit{grant}$, where busy is an input signal and grant is an output signal. The formula is inherently vacuous in the context of open systems, but is not inherently vacuous in the context of closed systems (considering the semantic-based definition of vacuity in both). To see this, consider the subformula $\psi = \mathsf{G}\,(\mathit{busy} \rightarrow \mathsf{F}\,(\mathit{grant} \wedge \neg \mathit{busy}))$ in φ. Since ψ imposes restrictions on the input signal, it is unrealizable. Hence, φ is o-equivalent to its ψ-strengthening, which is equivalent to $\mathsf{G}\,\mathsf{F}\,\mathit{grant}$. On the other hand, φ is not c-equivalent to any strengthening of it.

Tightening Type. The third parameter refers to the goal of the designer. In early stages of the design, the designer may be interested in detecting cases where the formula can be mutated to a formula that is strictly stronger, yet is still satisfiable (or, in the context of open systems, still realizable). The third parameter indicates whether the mutated formula has to be equivalent to the original formula or only has to maintain its satisfiability or realizability. Note that tightening of the specification to a specification that is strictly stronger and yet maintains its satisfiability or realizability is useful mainly in the context of synthesis, where it suggests that the specification for the system should be tightened. We demonstrate this in the two examples below. As Example 4 shows, there are formulas that can be mutated in a way that preserves realizability, yet cannot be mutated to an equivalent formula. Thus, inherent vacuity for such formulas is detected only with the third parameter indicating that we are looking for a mutation that maintains realizability.

Example 4. Consider the formula $\varphi = (\mathit{busy} \vee \mathit{ack}) \rightarrow \mathsf{X}\,\mathit{grant}$ where busy is an input signal and where ack and grant are output signals. There exists no sub-formula ψ of φ such that its ψ-strengthening is equivalent to φ. On the other hand, the ack-strengthening of φ, which is equivalent to $\varphi' = \mathit{busy} \rightarrow \mathsf{X}\,\mathit{grant}$, is realizable.

Example 5. Consider an open system with input req and outputs grant_1 and grant_2, and the formula $\varphi = \mathsf{G}\,(\mathit{req} \rightarrow \mathsf{X}\,(\mathit{grant}_1 \vee \mathit{grant}_2)) \wedge \mathsf{G}\,(\mathit{req} \rightarrow \mathsf{F}\,\mathit{grant}_2)$. The σ-strengthening of φ, for σ being the first occurrence of grant_2 is still realizable. Thus, the formula is inherently vacuous when the tightening type is "realizability preservation" (and the vacuity type is mutation of a single occurrence). The designer may want to tighten the unbounded delay in the second conjunct as not to overlap the first conjunct.

Consider the formula $\varphi' = \mathsf{G}\,(\mathit{req} \rightarrow \mathsf{F}\,(\mathit{grant}_1 \vee \mathit{grant}_2))$. The grant_2-strengthening of φ is $\mathsf{G}\,(\mathit{req} \rightarrow \mathsf{F}\,\mathit{grant}_1)$, which is still realizable. Thus, φ' is inherently vacuous according to the same criteria as above. Note that the same holds for the grant_1-strengthening of φ. In this case, however, it is not clear that the grant_2-strengthening or the grant_1-strengthening of φ are the desired formulas, as they both ignore a particular type of grant. The information from the check is still useful, as the specifier may conclude that the formula he has to use is $\mathsf{G}\,(\mathit{req} \rightarrow [\mathsf{F}\,\mathit{grant}_1 \wedge \mathsf{F}\,\mathit{grant}_2])$, which is the conjunction of the two strengthenings.

Polarity Type. In the context of model checking, a formula is checked for vacuity only after it has been verified to hold on the system. The vacuity check then examines whether a mutation that strengthens the formula still holds on the system Clearly, it makes no sense to model check a weaker formula, as it is guaranteed to be satisfied.[3] In the context of property-based design, however, we may consider mutations that strengthen the formula as well as mutations that weaken it. Indeed, in the extreme case a mutation that weakens the formula is the formula *true*, as all models satisfy it, and clearly a formula that is equivalent to *true* (yet is syntactically different) is inherently vacuous.

The fourth parameter in our framework refers to the polarity of the mutation, and indicates whether we compare the formula with its ψ-strengthening or ψ-weakening. The ψ-weakening of a formula φ is defined in a manner dual to its ψ-strengthening. For example, dualizing the definition of vacuity in [2], the ψ-weakening of φ is $\exists x.\varphi[\psi \leftarrow x]$, in which all the occurrences of ψ are replaced by an existentially quantified proposition. Likewise, dualizing the definition of vacuity in [20], the σ-weakening of φ, for an occurrence σ of some subformula, is $\varphi[\sigma \leftarrow \top]$, which is obtained from φ by replacing the occurrence σ by *true* if σ is of positive polarity and by *false* if σ is of negative polarity.

Example 6. Consider the formula $\varphi = (\mathsf{F}\, grant) \wedge (\mathsf{X}\, grant)$. Clearly, the formula φ is equivalent to its second conjunct, namely, $\mathsf{X}\, grant$, which is the $\mathsf{F}\, grant$-weakening of φ. On the other hand, there is no subformula of φ or an occurrence of a subformula ψ such that φ is equivalent to its ψ-strengthening.

Consider the formula $\varphi = (\mathsf{F}\, grant) \vee (\mathsf{X}\, grant)$. Since φ is equivalent to its $\mathsf{X}\, grant$-strengthening, namely, $\mathsf{F}\, grant$, we have that φ is inherently vacuous in a definition that considers strengthening. On the other hand, there exists no subformula of φ or an occurrence of a subformula ψ such that φ is equivalent to its ψ-weakening.

This shows there exist formulas that are inherently vacuous according to weakening but not according to strengthening and vice versa.

Example 7. In [9], the authors consider specifications of the form $\varphi = \bigwedge_{i\in I} \varphi_i$ and study how to detect redundant conjuncts. Formally, φ_j is redundant if φ is equivalent to $\bigwedge_{i\in I\setminus\{j\}} \varphi_i$. Note that this is a special case of our inherent vacuity when considering weakening of the mutation (and the syntactic-based definition of vacuity).

For example, the formula $\varphi = (wait\, \mathsf{U}\, busy) \wedge \mathsf{F}\,(wait \vee busy)$ is inherently vacuous according to this criterion. Indeed, φ is equivalent to $(wait\, \mathsf{U}\, busy)$, which is its σ-weakening, for $\sigma = \mathsf{F}\,(wait \vee busy)$.

3.2 Working with the Different Parameters

In order to describe the different parameters, we use the term φ is *inherently vacuous (by mutation) of type* $(\text{V}, \text{E}, \text{T}, \text{P})$, where $\text{V} \in \{s_{\text{V}}, m_{\text{V}}\}$ denotes the vacuity type (single or multiple occurrences), $\text{E} \in \{c_{\text{E}}, o_{\text{E}}\}$ denotes the equivalence type (closed or open systems), $\text{T} \in \{e_{\text{T}}, p_{\text{T}}\}$ denotes the tightening type (to an equivalent one or to one that preserves satisfaction), and $\text{P} \in \{s_{\text{P}}, w_{\text{P}}\}$ denotes the type of polarity (strengthening or weakening). For example,

[3] Nevertheless, [15] shows how one can benefit from checking the vacuity of negations of formulas that pass.

– φ is inherently vacuous of type $(m_{\mathrm{V}}, o_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$ if $\varphi \equiv_o \forall x.\varphi[\psi \leftarrow x]$, for some subformula ψ of φ.
– φ is inherently vacuous of type $(s_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, w_{\mathrm{P}})$ if $\varphi \equiv_c \varphi[\sigma \leftarrow \top]$, for some occurrence σ of a subformula of φ.
– φ is inherently vacuous of type $(m_{\mathrm{V}}, o_{\mathrm{E}}, p_{\mathrm{T}}, s_{\mathrm{P}})$ if $\forall x.\varphi[\psi \leftarrow x]$ is realizable for some subformula ψ of φ.

Note that inherent vacuity discussed in Section 2 is of type $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$. Note also that the parameters are orthogonal to each other. An exception is the tightening type and its polarity: if T is p_{T}, then P must be s_{P}.

For uniformity, we use the V, E, and T parameters also in other notations. In particular, a V-subformula of φ is a subformula if $\mathrm{V} = m_{\mathrm{V}}$ and is an occurrence of a subformula if $\mathrm{V} = s_{\mathrm{V}}$. Likewise, a E-system is a Kripke structure when $\mathrm{E} = c_{\mathrm{E}}$ and is a transducer when $\mathrm{E} = o_{\mathrm{E}}$. Finally, a formula that is E-satisfiable is satisfiable when $\mathrm{E} = c_{\mathrm{E}}$ and is realizable when $\mathrm{E} = o_{\mathrm{E}}$.

Example 8. Mutating a single occurrence of a subformula may not detect inherent vacuity that originates from the relations between different parts of the formula. For example, the formula $\varphi = (\mathit{wait} \wedge \mathit{busy}) \vee (\mathit{wait} \wedge \neg \mathit{busy})$ is inherently vacuous of types $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$ and $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, w_{\mathrm{P}})$ but is not inherently vacuous of type $(s_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$ or $(s_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, w_{\mathrm{P}})$. Indeed, the busy-strengthening of φ and the busy-weakening of it, which are equivalent to the formula wait, are equivalent to φ. However, there is no single occurrence σ of a subformula ψ such that $\varphi[\sigma \leftarrow \bot]$ or $\varphi[\sigma \leftarrow \top]$ is equivalent to wait.

On the other hand, mutating all occurrences may not detect local problems that are covered by other parts of the formula. For example, the formula $\varphi = (\mathsf{G}\, \mathit{high}) \vee (\mathit{high} \rightarrow \mathsf{F}\,\mathsf{G}\, \mathit{high})$ is inherently vacuous of type $(s_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$ but is not inherently vacuous of type $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$. Indeed, φ is equivalent to its σ-strengthening, for σ being the first occurrence of $\mathsf{G}\, \mathit{high}$. However, there is no subformula ψ of φ such that φ is equivalent to its ψ-strengthening.

Theorem 3 below summarizes the relations among the various types of inherent vacuity. The first implication follows from the implications of o-equivalence by c-equivalence [13]. The second implication follows from the fact we restrict attention to E-satisfiable formulas. Finally, all the incomparability results are demonstrated in the examples.

Theorem 3. *Let* $\mathrm{V} \in \{s_{\mathrm{V}}, m_{\mathrm{V}}\}$, $\mathrm{E} \in \{c_{\mathrm{E}}, o_{\mathrm{E}}\}$, $\mathrm{T} \in \{e_{\mathrm{T}}, p_{\mathrm{T}}\}$, *and* $\mathrm{P} \in \{s_{\mathrm{P}}, w_{\mathrm{P}}\}$.

1. *Inherent vacuity of type* $(\mathrm{V}, c_{\mathrm{E}}, \mathrm{T}, \mathrm{P})$ *implies inherent vacuity of type* $(\mathrm{V}, o_{\mathrm{E}}, \mathrm{T}, \mathrm{P})$. *Implication in the other direction does not hold.*
2. *For* E*-satisfiable formulas, inherent vacuity of type* $(\mathrm{V}, \mathrm{E}, e_{\mathrm{T}}, s_{\mathrm{P}})$ *implies inherent vacuity of type* $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$. *Implication in the other direction does not hold.*
3. *Inherent vacuity of type* $(m_{\mathrm{V}}, \mathrm{E}, \mathrm{T}, \mathrm{P})$ *is incomparable with inherent vacuity of type* $(s_{\mathrm{V}}, \mathrm{E}, \mathrm{T}, \mathrm{P})$.
4. *Inherent vacuity of type* $(\mathrm{V}, \mathrm{E}, e_{\mathrm{T}}, s_{\mathrm{P}})$ *is incomparable with inherent vacuity of type* $(\mathrm{V}, \mathrm{E}, e_{\mathrm{T}}, w_{\mathrm{P}})$.

We now turn to discuss the complexity of detecting inherent vacuity in the different settings. The following theorem shows that the problem of deciding whether a given LTL formula is inherently vacuous of various types we have defined, is not more difficult than the corresponding E-satisfiability problem for LTL.

Theorem 4. *Let* $\text{V} \in \{s_{\text{V}}, m_{\text{V}}\}$*,* $\text{E} \in \{c_{\text{E}}, o_{\text{E}}\}$*,* $\text{T} \in \{e_{\text{T}}, p_{\text{T}}\}$*, and* $\text{P} \in \{s_{\text{P}}, w_{\text{P}}\}$*. Given an* LTL *formula* φ *and a* V*-subformula* ψ *of* φ*, deciding whether* φ *is inherently vacuous of type* $(\text{V}, \text{E}, \text{T}, \text{P})$ *with witness* ψ *is PSPACE-complete for* $\text{E} = c_{\text{E}}$ *(except when* $\text{V} = m_{\text{V}}$ *and* $\text{T} = p_{\text{T}}$*, in which case it is in EXPSPACE-complete) and is 2EXPTIME-complete for* $\text{E} = o_{\text{E}}$*.*

Proof. For the upper bound, all cases with $\text{T} = e_{\text{T}}$ are reducible to checking the E-equivalence of φ and its mutation. When $\text{V} = m_{\text{V}}$, the mutation may involve universal or existential quantification of atomic propositions. Still, only one direction of the implication between φ and the mutation should be checked (the other direction always holds), and fortunately, it is the direction that can be reduced to LTL E-implication. The upper bounds then follow from the PSPACE and 2EXPTIME complexities for LTL closed and open implication, respectively [17, 29].

When $\text{T} = p_{\text{T}}$, we have to check whether the ψ-strengthening of φ is E-satisfiable. When $\text{V} = s_{\text{V}}$, the ψ-strengthening is an LTL formula, and again the upper bound follows from the known PSPACE and 2EXPTIME complexities for LTL E-implication. When $\text{V} = m_{\text{V}}$, the ψ-strengthening involves universal quantification of atomic propositions. When $\text{E} = c_{\text{E}}$, the problem reduces to the satisfiability problem of LTL augmented with universal quantification over atomic propositions, this leads to an EXPSPACE complexity [32]. When $\text{E} = o_{\text{E}}$, we can check in 2EXPTIME the realizability of $\neg\varphi[\psi \leftarrow x]$ in a dual setting. By the determinacy of realizability, the latter is realizable iff $\forall x.\varphi[\psi \leftarrow x]$ is unrealizable.

For the lower bound, taking ψ to be φ reduces E-satisfiability to inherent vacuity, thus the lower bound holds from the known PSPACE-hardness and 2EXPTIME-hardness for LTL satisfiability and realizability, respectively [26, 29]. An exception is inherent vacuity of type $(m_{\text{V}}, c_{\text{E}}, p_{\text{T}}, s_{\text{P}})$, to which we reduce satisfiability of LTL augmented with universal quantification over atomic propositions, which is known to be EXPSPACE-hard [32]. □

Corollary 2. *Let* $\text{V} \in \{s_{\text{V}}, m_{\text{V}}\}$*,* $\text{T} \in \{e_{\text{T}}, p_{\text{T}}\}$*, and* $\text{P} \in \{s_{\text{P}}, w_{\text{P}}\}$*. The problem of deciding whether an* LTL *formula is inherently vacuous of type* $(\text{V}, c_{\text{E}}, \text{T}, \text{P})$ *can be solved in polynomial space (except for type* $(m_{\text{V}}, c_{\text{E}}, p_{\text{T}}, s_{\text{P}})$*, which requires exponential space). The problem of deciding whether an* LTL *formula is inherently vacuous of types* $(\text{V}, o_{\text{E}}, \text{T}, \text{P})$ *can be solved in doubly exponential time.*

Note that Theorem 2 and Corollary 1 are a special case of Theorem 4 and Corollary 2.

Corollary 2 shows that detection of inherent vacuity, while being more informative than detection of satisfiability or realizability, which are used in property-based design [28], is not harder than these basic problems.[4]

[4] The exception of $(m_{\text{V}}, c_{\text{E}}, p_{\text{T}}, s_{\text{P}})$ follows from the universal quantification of atomic propositions that vacuity type m_{V} involves. It suggests that designers that suspect their specification for closed systems should be tightened may prefer to work with vacuity type s_{V}.

Having refined the notion of inherent vacuity by mutations, we now turn to refine the alternative approach of inherent vacuity by model. Note that since the definition of inherent vacuity by model refers to vacuous satisfaction of φ in a model that satisfies φ, it is not interesting to consider weakening of formulas. Thus, when we compare the notions of inherent vacuity by mutation and by model, the fourth parameter has to be s_{P}.

Definition 3. *Consider an* LTL *formula* φ. *For* $\mathrm{V} \in \{s_{\mathrm{V}}, m_{\mathrm{V}}\}$, $\mathrm{E} \in \{c_{\mathrm{E}}, o_{\mathrm{E}}\}$, *and* $\mathrm{T} \in \{e_{\mathrm{T}}, p_{\mathrm{T}}\}$, *we say that*

– *φ is inherently vacuous by model of type $(\mathrm{V}, \mathrm{E}, e_{\mathrm{T}}, s_{\mathrm{P}})$ if φ is satisfied vacuously in all* E*-systems that satisfy φ.*
– *φ is inherently vacuous by model of type $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$ if φ is satisfied vacuously in some* E*-system that satisfies φ.*

Note that for type $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$, the definition coincides with Definition 2.

The following theorem, extending Theorem 1, states that the two approaches for inherent vacuity coincide all over the framework.

Theorem 5. *For all $\mathrm{V} \in \{s_{\mathrm{V}}, m_{\mathrm{V}}\}$, $\mathrm{E} \in \{c_{\mathrm{E}}, o_{\mathrm{E}}\}$, and $\mathrm{T} \in \{e_{\mathrm{T}}, p_{\mathrm{T}}\}$, an* LTL *formula φ is inherently vacuous by mutation of type $(\mathrm{V}, \mathrm{E}, \mathrm{T}, s_{\mathrm{P}})$ iff φ is inherently vacuous by model of type $(\mathrm{V}, \mathrm{E}, \mathrm{T}, s_{\mathrm{P}})$.*

Proof. Theorem 1 provides the proof for type $(m_{\mathrm{V}}, c_{\mathrm{E}}, e_{\mathrm{T}}, s_{\mathrm{P}})$. The proof for the other types with $\mathrm{T} = e$ are similar. In fact, in the setting of open systems and transducers, the equivalence is valid even when we consider deterministic transducers. To see this, note that the "only if" direction in the proof of Theorem 1 is based on defining the union of k Kripke structures as a single Kripke structure. While this cannot be done with deterministic Kripke structures, it can be done with deterministic transducers. Indeed, by adding $\lceil \log k \rceil$ input signals that do not appear in the formula, we can define a deterministic initial transition function for the union, in which the added input signals choose a transducer from the union. The rest of the proof is the same as in the case of nondeterministic Kripke structures.

It is left to describe the details for the case $\mathrm{T} = p$, which is different.

Assume that φ is inherently vacuous by mutation of type $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$. Let ψ be such that the ψ-strengthening of φ is E-satisfiable, and let $\mathcal{S}$ be the E-system that satisfies it. By definition, ψ does not affect the satisfaction of φ in $\mathcal{S}$, thus $\mathcal{S}$ satisfies φ vacuously, and φ is inherently vacuous by model of type $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$.

For the other direction, if φ is inherently vacuous by model of type $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$, then there exists a V-subformula ψ that does not affect its satisfaction in some E-system. Then the ψ-strengthening of φ is E-satisfiable, thus φ is inherently vacuous by mutation of type $(\mathrm{V}, \mathrm{E}, p_{\mathrm{T}}, s_{\mathrm{P}})$. $\square$

4 Discussion

We proposed a framework for inherent vacuity — vacuity of specifications without a reference model. We argue that, as has been the case with vacuity in model checking, inherent vacuity is common, and detection of inherent vacuity may significantly improve the specifications and the designer's understanding of it.

In [9], the authors experimented with a real-life block as described in [24]. Its specification consists of 50 formulas. It is shown in [9] that inherent vacuity exists already with the basic definition of redundant conjuncts. Indeed, in a set consisting of 17 formulas, 9 were found to be redundant. Another common source of inherent vacuity, not captured by the definition in [9], is the fact that subformulas that appear in different conjunctions may be related, without the specifier being aware of such a relation. In particular, a formula for the full specification may be written by a group of specifiers, each specifying a different aspect of the design. For example, consider the specification $\varphi = \varphi_1 \wedge \varphi_2 \wedge \vartheta$ where φ_1 is $\mathsf{G}\,(\xi_1 \rightarrow \mathsf{F}\,\psi)$, φ_2 is $\mathsf{G}\,(\xi_2 \rightarrow \mathsf{F}\,\psi)$, and ϑ is $\mathsf{G}\,(\xi_1 \vee \xi_2)$. Such a specification is classical in the sense that ξ_1 and ξ_2 represent some modes of operation, and ϑ states that the system is always in one of the modes. The formula φ is inherently vacuous as it is equivalent to $\mathsf{G}\,(\xi_1 \vee \xi_2) \wedge \mathsf{G}\,\mathsf{F}\,\psi$. Yet, as different specifiers may have specified φ_1 and φ_2, such a vacuity may not be noticed.

The above phenomenon, of subformulas that are related to each other without the specifier being aware of it, follows from the fact that specifiers often pack complicated properties into a single temporal formula. Moreover, today standard temporal logics (e.g. SVA [33], PSL [12, 25],) provide a mechanism for doing so, by allowing one to name a formula and then relate to it in other formulas. As the referenced subformulas may have many signals in common, inherent vacuity in the obtained formula is likely to occur.

Another reason for finding inherent vacuity is mistakes, either typos or small logical errors done by novice in temporal logic. For example, a typo in the formula $(p \rightarrow (\varphi \wedge \psi)) \ \wedge(\neg p \rightarrow (\varphi \wedge \vartheta))$ can result in the formula $(p \rightarrow (\varphi \wedge \psi)) \wedge (p \rightarrow (\varphi \wedge \vartheta))$, which is inherently vacuous by mutation replacing the second occurrence of φ by *true*. As another example, trying to write the temporal-logic formula for the English specification "If signal *error* is asserted, it will remain asserted forever" a novice might write $\varphi = \mathsf{G}\,(\mathit{error} \rightarrow (\mathit{error}\,\mathsf{U}\,\mathit{error}))$ which is a tautology, and inherently vacuous by the mutation $\forall x.\varphi(\mathit{error} \leftarrow x)$. Similarly, consider the English specification "If signal *grant* is not asserted the cycle after a *req*uest, then it cannot be asserted two cycles after the request." A wrong attempt to formalize it may be $\mathsf{G}\,\neg(\mathit{req} \rightarrow \mathsf{X}\,(\neg \mathit{grant} \rightarrow \mathsf{X}\,(\mathit{grant})))$. The latter formula is inherently vacuous by the mutation replacing the second occurrence of *grant* by *false*. Similar examples abound.

We note that inherent vacuity does not always imply that the specification should be changed to its simpler mutation, and sometimes the contribution of detecting inherent vacuity is a better understanding of the specification, possibly leading to a change in the specification that is different from replacing it by the mutated specification. To see this, let us consider again the specification $\varphi = \mathsf{G}\,(\xi_1 \rightarrow \mathsf{F}\,\psi) \wedge \mathsf{G}\,(\xi_2 \rightarrow \mathsf{F}\,\psi) \wedge \mathsf{G}\,(\xi_1 \vee \xi_2)$ discussed above. While φ is equivalent to $\mathsf{G}\,(\xi_1 \vee \xi_2) \wedge \mathsf{G}\,\mathsf{F}\,\psi$, the specifier may prefer to leave the original formula, in case the formula needs to be refined further in later stages of the design (and different refinements may be needed for the different models ξ_i), or in case the assumption that only these two modes are possible is removed. Still, it is useful information for the designer to know that, as is, ψ happens infinitely often, regardless of the mode. Note, however, that in synthesis one would always prefer the simpler mutated specification as it will induce a smaller system in less time.

Acknowledgments

We thank Shoham Ben-David, Hana Chockler, Cindy Eisner and Avigail Orni for helpful comments on an early draft of this paper and for interesting discussions on the subject.

References

1. Ammons, G., Mandelin, D., Bodk, R., Larus, J.R.: Debugging temporal specifications with concept analysis. In: Proc. PLDI 2003, pp. 182–195 (2003)
2. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.Y.: Enhanced vacuity detection for linear temporal logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 368–380. Springer, Heidelberg (2003)
3. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. FMSD 18(2), 141–162 (2001)
4. Ben-David, S., Fisman, D., Ruah, S.: Temporal antecedent failure: Refining vacuity. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 492–506. Springer, Heidelberg (2007)
5. Bloem, R., Cavada, R., Pill, I., Roveri, M., Tchaltsev, A.: RAT: A tool for the formal analysis of requirements. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 263–267. Springer, Heidelberg (2007)
6. Bustan, D., Flaisher, A., Grumberg, O., Kupferman, O., Vardi, M.Y.: Regular vacuity. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 191–206. Springer, Heidelberg (2005)
7. Chechik, M., Gheorghiu, M., Gurfinkel, A.: Finding state solutions to temporal queries. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 273–292. Springer, Heidelberg (2007)
8. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for temporal logic model checking. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 528–542. Springer, Heidelberg (2001)
9. Chockler, H., Strichman, O.: Easier and more informative vacuity checks. In: Proc. 5th MEMOCODE, pp. 189–198 (2007)
10. Claessen, K.: A coverage analysis for safety property lists. In: 32nd MFCS, pp. 139–145. IEEE Computer Society, Los Alamitos (2007)
11. Dong, Y., Sarna-Starosta, B., Ramakrishnan, C.R., Smolka, S.A.: Vacuity checking in the modal μ-calculus. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 147–162. Springer, Heidelberg (2002)
12. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
13. Greimel, K., Bloem, R., Jobstmann, B., Vardi, M.: Open Implication. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 361–372. Springer, Heidelberg (2008)
14. Gurfinkel, A., Chechik, M.: Extending extended vacuity. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 306–321. Springer, Heidelberg (2004)
15. Gurfinkel, A., Chechik, M.: How vacuous is vacuous. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 451–466. Springer, Heidelberg (2004)
16. Hoskote, Y., Kam, T., Ho, P.-H., Zhao, X.: Coverage estimation for symbolic model checking. In: Proc. 36th DAC, pp. 300–305 (1999)
17. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)

18. Kupferman, O.: Sanity checks in formal verification. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 37–51. Springer, Heidelberg (2006)
19. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. FMSD 19(3), 291–314 (2001)
20. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. STTT 4(2), 224–233 (2003)
21. Namjoshi, K.S.: An efficiently checkable, proof-based formulation of vacuity in model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 57–69. Springer, Heidelberg (2004)
22. Oberkönig, M., Schickel, M., Eveking, H.: A quantitative completeness analysis for property-sets. In: Proc. 7th FMCAD, pp. 158–161. IEEE Computer Society, Los Alamitos (2007)
23. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th POPL, pp. 179–190 (1989)
24. PROSYD. The Prosyd project on property-based system design (2007), http://www.prosyd.org
25. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850$^{\text{TM}}$ (2005)
26. Rosner, R.: Modular Synthesis of Reactive Systems. Ph.D thesis, Weizmann Institute of Science (1992)
27. Roveri, M.: Novel techniques for property assurance. Technical report, PROSYD FP6-IST-507219 (2007)
28. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
29. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. Journal of the ACM 32, 733–749 (1985)
30. Samer, M., Veith, H.: Parametrized vacuity. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 322–336. Springer, Heidelberg (2004)
31. Samer, M., Veith, H.: On the notion of vacuous truth. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 2–14. Springer, Heidelberg (2007)
32. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. In: TCS, vol. 49, pp. 217–237 (1987)
33. Annex E of IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800$^{\text{TM}}$ (2005)
34. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. I& C 115(1), 1–37 (1994)

A Meta Heuristic for Effectively Detecting Concurrency Errors

Neha Rungta and Eric G. Mercer

Department of Computer Science
Brigham Young University
Provo, UT 84602, USA

Abstract. Mainstream programming is migrating to concurrent architectures to improve performance and facilitate more complex computation. The state of the art static analysis tools for detecting concurrency errors are imprecise, generate a large number of false error warnings, and require manual verification of each warning. In this paper we present a meta heuristic to help reduce the manual effort required in the verification of warnings generated by static analysis tools. We manually generate a small sequence of program locations that represent points of interest in checking the feasibility of a particular static analysis warning; then we use a meta heuristic to automatically control scheduling decisions in a model checker to guide the program along the input sequence to test the feasibility of the warning. The meta heuristic guides a greedy depth-first search based on a two-tier ranking system where the first tier considers the number of program locations already observed from the input sequence, and the second tier considers the perceived closeness to the next location in the input sequence. The error traces generated by this technique are real and require no further manual verification. We show the effectiveness of our approach by detecting feasible concurrency errors in benchmarked concurrent programs and the JDK 1.4 concurrent libraries based on warnings generated by the Jlint static analysis tool.

1 Introduction

The ubiquity of multi-core Intel and AMD processors is prompting a shift in the programming paradigm from inherently sequential programs to concurrent programs to better utilize the computation power of the processors. Although parallel programming is well studied in academia, research, and a few specialized problem domains, it is not a paradigm commonly known in mainstream programming. As a result, there are few, if any, tools available to programmers to help them test and analyze concurrent programs for correctness.

Static analysis tools that analyze the source of the program for detecting concurrency errors are imprecise and incomplete [1,2,3,4]. Static analysis techniques are not always useful as they report warnings about errors that *may* exist in the program. The programmer has to manually verify the feasibility of the warning by reasoning about input values, thread schedules, and branch conditions required to manifest the error along a real execution path in the program. Such manual verification is not tractable in mainstream software development because of the complexity and the cost associated with such an activity.

Model checking in contrast to static analysis is a precise, sound, and complete analysis technique that reports only feasible errors [5,6]. It accomplishes this by exhaustively enumerating all possible behaviors (state space) of the program to check for the presence and absence of errors; however, the growing complexity of concurrent systems leads to an exponential growth in the size of state space. This state space explosion has prevented the use of model checking in mainstream test frameworks.

Directed model checking focuses its efforts in searching parts of the state space where an error is more likely to exist in order to partially mitigate the state space explosion problem [7,8,9,10,11]. Directed model checking uses heuristic values and path-cost to rank the states in order of interest in a priority queue. Directed model checking uses some information about the program or the property being verified to generate heuristic values. The information is either specified by the user or computed automatically. In this work we use the imprecise static analysis warnings to detect possible defects in the program and use a precise directed search with a meta heuristic to localize real errors.

The meta heuristic presented in this paper guides the program execution in a greedy depth-first manner along an input sequence of program locations. The input sequence is a small number of locations manually generated such that they are relevant in testing the feasibility of a static analysis warning or a reachability property. The meta heuristic ranks the states based on the portion of the input sequence already observed. States that have observed a greater number of locations from the input sequence are ranked as more *interesting* compared to other states. In the case where multiple states have observed the same number of locations in the sequence, the meta heuristic uses a secondary heuristic to guide the search toward the next location in the sequence. In essence, the meta heuristic automatically controls scheduling decisions to drive the program execution along the input sequence in a greedy depth-first manner. The greedy depth-first search picks the best-ranked immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack.

In this work we do not consider any non-determinism arising due to data input and only consider the non-determinism arising from thread schedules. The error traces generated by the technique are real and require no further verification; however, if the technique does not find an error we cannot prove the absence of the error. The technique is sound in error detection but not complete.

To test the validity of our meta heuristic solution in aiding the process of automatically verifying deadlocks, race conditions, and reachability properties in multi-threaded programs, we present an empirical study conducted on several benchmarked concurrent Java programs and the JDK 1.4 concurrent libraries. We use the Java PathFinder model checker (an explicit state Java byte-code model checker) to conduct the empirical study [6]. We show that the meta heuristic is extremely effective in localizing a feasible error when given a few key locations relevant to a corresponding static analysis warning. Furthermore, the results demonstrate that the choice of the secondary heuristic has a dramatic effect on the number of states generated, on average, before error discovery.

2 Meta Heuristic

In this section we describe the input sequence to the meta heuristic, our greedy depth-first search, and the guidance strategy based on the meta heuristic.

2.1 Input Sequence

The input to our meta heuristic is the program, an environment that closes the program, and a sequence of locations that are relevant to checking the feasibility of the static analysis warning. The number and type of locations in the sequence can vary based on the static analysis warning being verified. For example, to test the occurrences of race-conditions, we can generate a sequence of program locations that represent a series of reads and writes on shared objects. Note that we do not manually specify which thread is required to be at a given location in the input sequence and rely on the meta heuristic to intelligently pick thread assignments.

We use the example in Fig. 1 to demonstrate how we generate an input sequence to check the feasibility of a possible race condition from a static analysis warning. Fig. 1 represents a portion of a program that uses the JDK 1.4 concurrent public library. The `raceCondition` class in Fig. 1(a) initializes two `AbstractList` data structures, l_1 and l_2, using the synchronized `Vector` sub-class implementation. Two threads of type `AThread`, t_0 and t_1, are initialized such that both threads can concurrently access and modify the data structures, l_1 and l_2. Finally, `main` invokes the `run` function of Fig. 1(b) on the two threads. The threads go through a sequence of events, including operations on l_1 and l_2 in Fig. 1(b). Specifically, an `add` operation is performed on list l_2 when a certain condition is satisfied; the add is then followed by an operation that checks whether l_1 `equals` l_2. The `add` operation in the `Vector` class, Fig. 1(c), first acquires a lock on its own `Vector` instance and then adds the input element to the instance. The `equals` function in the same class, however, acquires the lock on its own instance and invokes the `equals` function of its parent class which is `AbstractList` shown in Fig. 1(d).

The Jlint static analysis tool issues a series of warnings about potential concurrency errors in the concurrent JDK library when we analyze the program shown in Fig. 1 [4]. The Jlint warnings for the `equals` function in the `AbstractList` class in Fig. 1(d) are on the Iterator operations (lines $8 - 14$ and lines $18 - 19$). The warnings state that the Iterator operations are not synchronized. As the program uses a synchronized `Vector` sub-class of the `AbstractList` (in accordance with the specified usage documentation), the user may be tempted to believe that the warnings are spurious. Furthermore, people most often ignore the warnings in libraries since they assume the libraries to be error-free.

```
1: class raceCondition{
2: ...
3:   public static void main(){
4:     AbstractList $l_1$ := new Vector();
5:     AbstractList $l_2$ := new Vector();
6:     AThread $t_0$ = new AThread($l_1, l_2$);
7:     AThread $t_1$ = new AThread($l_1, l_2$);
8:     $t_0$.start(); $t_1$.start();
9:     ...
10:  }
11: ...
12: }
```

(a)

```
1: class AThread extends Thread{
2:   AbstractList $l_1$;
3:   AbstractList $l_2$;
4:   AThread(AbstractList $l_1$,
5:                AbstractList $l_2$){
6:     this.$l_1$ := $l_1$; this.$l_2$ := $l_2$;
7:   }
8: public void run(){
9:     ...
10:     if some_condition then
11:       $l_2$.add(some_object);
12:     ...
13:     $l_1$.equals($l_2$);
14:     ...
15:   }
16: }
```

(b)

```
1: class Vector extends
2:                   AbstractList{
3: ...
4:   public synchronized boolean equals
5:                         (Object $o$){
6:     super.equals($o$);
7:   }
8: ...
9:   public synchronized boolean add
10:                        (Object $o$){
11:     $modCnt{+}{+}$;
12:     ensureCapacityHelper($cnt + 1$);
13:     elementData[$cnt{+}{+}$] = $o$;
14:     return true;
15:   }
16: ...
17: }
```

(c)

```
1:   class AbstractList
2:                  implements List{
3:   public boolean equals(Object $o$){
4:     if $o$ == this then
5:       return true;
6:     if ¬($o$ instanceof List) then
7:       return false;
8:     ListIterator $e_1$ := ListIterator();
9:     ListIterator $e_2$ :=
10:                   (List $o$).listIterator();
11:    while $e_1$.hasNext() and
12:                   $e_2$.hasNext() do
13:       Object $o_1$ := $e_1$.next();
14:       Object $o_2$ := $e_2$.next();
15:       if¬($o_1$ == null ? $o_2$ == null :
16:                   $o_1$.equals($o_2$)) then
17:         return false;
18:    return ¬($e_1$.hasNext() ||
19:                   $e_2$.hasNext())
20:   }
21: }
```

(d)

Fig. 1. Possible race-condition in the JDK 1.4 concurrent library

To check the feasibility of the possible race condition reported by Jlint for the example in Fig. 1 we need a thread iterating over the list, l_2, in the `equals` function of `AbstractList` while another thread calls the `add` function. A potential input sequence of locations to test the feasibility of the warning is as follows:

1. Get the ListIterator, e_2 at lines $9 - 10$ in Fig. 1(d).
2. Check e_2 `hasNext`() at line 12 in Fig. 1(d).
3. Add `some_object` to l_2 at line 11 in Fig. 1(b).
4. Call e_2.`next`() at line 14 in Fig. 1(d).

```
1: /* $backtrack := \emptyset, visited := \emptyset$ */
procedure gdf_search($\langle s, locs, h_{val} \rangle$)
2: $visited := visited \cup \{s\}$
3: while $s \neq$ null do
4:    if error($s$) then
5:       report error statistics
6:       exit
7:    $\langle s, locs, h_{val} \rangle$ := choose_best_successor($\langle s, locs, h_{val} \rangle$)
8:    if $s ==$ null then
9:       $\langle s, locs, h_{val} \rangle$ := get_backtrack_state()
```

Fig. 2. Pseudocode for the greedy depth-first search

The same approach can be applied to generate input sequences for different warnings. Classic lockset analysis techniques detect potential deadlocks in multi-threaded programs caused due to cyclic lock dependencies [2,12]. For example, it detects a cyclic dependency in the series of lock acquisitions $l_0(A) \rightarrow l_1(B)$ and $l_9(B) \rightarrow l_{18}(A)$, where A and B are the locks acquired at different program locations specified by l_n. To generate an input sequence that checks the feasibility of the possible deadlock we can generate a sequence of locations: $l_0 \rightarrow l_9 \rightarrow l_1 \rightarrow l_{18}$. A larger set of concurrency error patterns are described by Farchi *et. al* in [13]. Understanding and recognizing the concurrent error patterns can be helpful in generating location sequences to detect particular errors.

In general, providing as much relevant information as possible in the sequence enables the meta heuristic to be more effective in defect detection; however, only 2–3 key locations were required to find errors in most of the models in our study. Any program location that we think affects the potential error can be added to the sequence. For example, if there is a data definition in the program that affects the variables in the predicate, `some_condition`, of the branch statement shown on line 10 in Fig. 1(b), then we can add the program location of the data definition to the sequence. Similarly we can generate input sequences to check reachability properties such as NULL pointer exceptions and assertion violations in multi-threaded programs.

2.2 Greedy Depth-First Search

In this subsection we describe a greedy depth-first search that lends itself naturally in directing the search using the meta heuristic along a particular path (the input sequence of locations). The greedy depth-first search mimics a test-like paradigm for multi-threaded programs. The meta heuristic can be also used with bounded priority-queue based best-first searches with comparable results.

The pseudocode for the greedy depth-first search is presented in Fig. 2. The input to gdf_search is a tuple with the initial state of the program (s), the sequence of locations ($locs$), and the initial secondary heuristic value (h_{val}). In a loop we guide the execution as long as the current state, s, has successors (lines $3-9$). At every state we check whether the state, s, satisfies the error condition (line 4). If an error is detected, then we report the error details and exit the search;

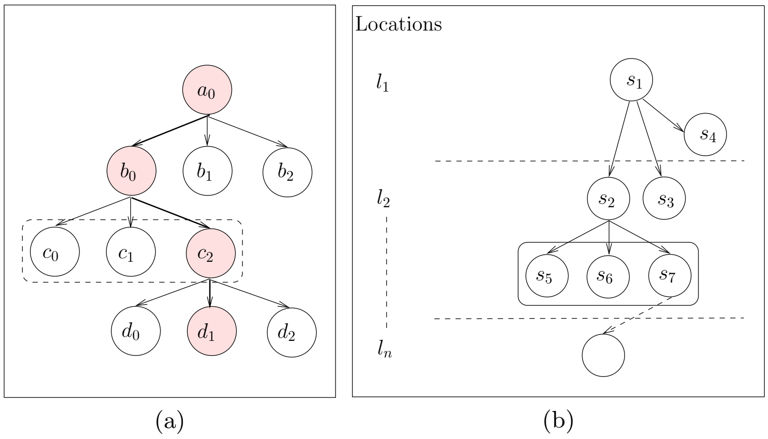


Fig. 3. Guidance (a) Greedy depth-first search (b) Two-level ranking scheme

otherwise, we continue to guide the search. The `choose_best_successor` function only considers the immediate successors of s and assigns to the current state the best-ranked successor of s (line 7). When the search reaches a state with no immediate successors, the technique requests a backtrack state as shown on lines $8-9$ in Fig. 2. The details of `choose_best_successor` and `get_backtrack_state` are provided in Fig. 4 and Fig. 5 respectively.

Fig. 3(a) demonstrates the greedy depth-first search using a simple example. The `choose_best_successor` function ranks c_0, c_1, and c_2 (enclosed in a dashed box) to choose the best successor of b_0 in Fig. 3(a). The shaded state c_2 is ranked as the best successor of b_0. When the search reaches state d_2 that does not have any successors, the search backtracks to one of the unshaded states (e.g., b_1, b_2, c_0, c_1, d_0, or d_2). We bound the number of unshaded states (backtrack states) saved during the search. Bounding the backtrack states makes our technique incomplete; although, the bounding is not a limitation because obtaining a complete coverage of the programs we are considering is not possible.

2.3 Guidance Strategy

The meta heuristic uses a two-tier ranking scheme as the guidance strategy. The states are first assigned a rank based on the number of locations in the input sequence that have been encountered along the current execution path. The meta heuristic then uses a secondary heuristic to rank states that observed the same number of locations in the sequence. The secondary heuristic is essentially used to guide the search toward the next location in the input sequence.

In Fig. 4 we present the pseudocode to choose the best successor of a given state. The input to the function is a tuple $\langle s, locs, h_{val} \rangle$ where s is a program state, $locs$ is a sequence of locations, and h_{val} is the heuristic value of s generated by the secondary heuristic function. We evaluate each successor of s, s', and

```
 1: /* $mStates := \emptyset$, $hStates := \emptyset$, $min\_h_{val} := \infty$ */
procedure choose_best_successor($\langle s, locs, h_{val} \rangle$)
 2: for each $s' \in$ successors($s$) do
 3:    if $\neg visited$.contains($s'$) then
 4:       $visited$.add_state($s'$)
 5:       $locs' := locs$ /* Make copy of locs */
 6:       $h'_{val}$ = get_h_value($s'$)
 7:       if $s'$.current_loc() == $locs$.top() then
 8:          $mStates$ := next_state_to_explore($mStates$, $\langle s', locs'$.pop(), $h'_{val} \rangle$)
 9:       else
10:          $hStates$ := next_state_to_explore($hStates$, $\langle s', locs, h'_{val} \rangle$)
11:       $backtrack$.add_state($\langle s', locs', h'_{val} \rangle$)
12: if $mStates \neq \emptyset$ then
13:    $\langle s, locs, h_{val} \rangle$ := get_random_element($mStates$)
14: else
15:    $\langle s, locs, h_{val} \rangle$ := get_random_element($hStates$)
16: $backtrack$.remove_state($\langle s, locs, h_{val} \rangle$)
17: bound_size($backtrack$)
18: return $\langle s, locs, h_{val} \rangle$

procedure next_state_to_explore($states$, $\langle s, locs, h_{val} \rangle$)
 1: if $states == \emptyset$ or $h_{val} == min\_h_{val}$ then
 2:    $states$.add_state($\langle s, locs, h_{val} \rangle$)
 3: else if $h_{val} < min\_h_{val}$ then
 4:    $states$.clear()
 5:    $states$.add_state($\langle s, locs, h_{val} \rangle$)
 6:    $min\_h_{val} := h_{val}$
 7: return $states$
```

Fig. 4. Two-tier ranking scheme for the meta heuristic

process s' if it is not found in the `visited` set (line $2-3$). To process s' we add it to the `visited` set (line 4), copy the sequence of locations *locs* into a new sequence of locations $locs'$ (line 5), and compute the secondary heuristic value for s' (line 6). If s' observes an additional location from the sequence (line 7), then we update the `mStates` set (line 8); otherwise, we update the `hStates` set (line 10). An element from the $locs'$ is removed on line 8 to indicate s' has observed an additional location. We invoke the `next_state_to_explore` function with the `mStates` or the `hStates` set and the tuple containing s'. The best successor is picked from `mStates` if it is non-empty; else, it is picked from the `hStates` set. The algorithm prefers states in the `mStates` set because they have observed an additional location compared to their parent. All other successor states are added to the `backtrack` set (lines $12-18$).

The `next_state_to_explore` function in Fig. 4 uses the secondary heuristic values (h_{val}) to add states to the `mStates` and `hStates` sets. Recall that the `next_state_to_explore` is invoked with either the `mStates` set or `hStates` set which is mapped to the formal parameter `states`. When the `states` set is empty or the h_{val} is equal to the minimum heuristic value (min_h_{val}) then the algorithm simply adds the tuple with the successor state to the `states` set. If, however, the h_{val} is less than the minimum heuristic value then the algorithm clears the `states` set, adds the tuple with the successor state to `states`, and sets the value of min_h_{val} to h_{val}. Finally, the function returns the `states` set.

```
procedure get_backtrack_state()
1: if $backtrack == \emptyset$ then
2:     return $\langle$null, $\infty$, $\infty\rangle$
3: else
4:     $x :=$ pick_backtrack_meta_level()
5:     $b\_points :=$ get_states($backtrack, x$)
6:     $b\_points := b\_points \cap$ states_min_h_value($b\_points$)
7:     return get_random_element($b\_points$)
```

Fig. 5. Stochastic backtracking technique

We use Fig. 3(b) to demonstrate the two-tier ranking scheme. In Fig. 3(b) the search is guided through locations l_1 to l_n. The dashed-lines separate the states based on the number of locations from the sequence they have observed along the path from the initial state. The states at the topmost level l_1 have encountered the first program location in the sequence while states at l_2 have observed the first two program locations from the sequence, so on and so forth. In Fig. 3(b) we see that state s_1 has three successors: s_2, s_3, and s_4. The states s_2 and s_3 observe an additional location, l_2, from the sequence compared to their parent s_1. Suppose s_2 and s_3 have the same secondary heuristic value. We add the states s_2 and s_3 to the `mStates` set to denote that a location from the sequence is observed. Suppose, the secondary heuristic value of s_4 is greater than that of s_2 and s_3; then s_4 is not added to the `hStates` set.

After enumerating the successors of s_1, the `mStates` set is non-empty so we randomly choose between s_2 and s_3 (line 13 in Fig. 4) and return the state as the best successor. When we evaluate successors of a state that do not encounter any additional location from the sequence, for example, the successors of s_2 in Fig. 3(b) (enclosed by the box), the states are ranked simply based on their secondary heuristic values. The best successor is then picked from the `hStates` set. All states other than the best successor are added to the `backtrack` set. We bound the size of the `backtrack` set to mitigate the common problem in directed model checking where saving the frontier in a priority queue consumes all memory resources.

The `get_backtrack_state` function in Fig. 5 picks a backtrack point when the guided test reaches the end of a path. Backtracking allows the meta heuristic to pick a different set of threads when it is unable to find an error along the initial sequence of thread schedules. As shown in Fig. 5, if the `backtrack` set is empty, then the function returns `null` as the next state (lines $1-2$); otherwise, the function probabilistically picks a meta level, x, between 1 and n where n is the number of locations in the sequence. The states that have observed one program location from the sequence are at meta level one. We then get all the states at meta level x and return the state with the minimum secondary heuristic value among the states at that meta level. The stochastic element of picking backtrack points enables the search to avoid getting stuck in a local minima.

3 Empirical Study

The empirical study in this paper is designed to evaluate the effectiveness of the meta heuristic in detecting concurrency errors in multi-threaded Java programs.

3.1 Study Design

We conduct the experiments on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of greedy depth-first search and randomized depth-first search. All the trials are bounded at one hour. We execute multiple trials of the greedy depth-first search since all ties in heuristic values are brokenly randomly and there is a stochastic element in picking backtrack points. An extensive study shows that randomly breaking ties in heuristic values helps in overcoming the limitations (and benefits) of default search order in directed search techniques [14]. We pick the time bound and number of trials to be consistent with other recent empirical studies [15,16,17]. Since each trial is completely independent of the other trials we use a super computing cluster of 618 nodes to distribute the trials on various nodes and quickly generate the results.[1] We use the Java Pathfinder (JPF) v4.0 Java byte-code model checker with partial order reduction turned on to run the experiments [6]. In the greedy depth-first search trials we save at most 100,000 backtrack states.

We use six unique multi-threaded Java programs in this study to evaluate the effectiveness of the meta heuristic in checking whether the input sequence leads to an error. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM Research Lab in Haifa, and classical concurrency errors described in literature [15]. We pick these three artifacts from the benchmark suite because the threads in these programs can be systematically manipulated to create configurations of the model where randomized depth-first search is unable to find errors in the models [17]. These models also exhibit different concurrency error patterns described by Farchi *et. al* in [13]. The other three examples are programs that use the JDK 1.4 library in accordance with the documentation. Fig. 1 is one such program that appears as `AbsList` in our results. We use Jlint on these models to automatically generate warnings on possible concurrency errors in the JDK 1.4 library and then manually generate the input sequences. The name, type of model, number of locations in the input sequence, and source lines of code (SLOC) for the models are as follows:

- **TwoStage:** Benchmark, Num of locs: 2, SLOC: 52
- **Reorder:** Benchmark, Num of locs: 2, SLOC: 44
- **Wronglock:** Benchmark, Num of locs: 3, SLOC: 38
- **AbsList:** Real, Num of locs: 6, Race-condition in the `AbstractList` class using the synchronized `Vector` sub-class Fig. 1. SLOC: 7267
- **AryList:** Real, Num of locs: 6, Race-condition in the `ArrayList` class using the synchronized `List` implementation. SLOC: 7169

[1] We thank Mary and Ira Lou Fulton for their generous donations to the BYU Supercomputing laboratory.

Table 1. Error density of the models with different search techniques

Subject	Total	Random	Meta Heuristic		
	Threads	DFS	PFSM	Rand	Prefer Threads
TwoStage(7,1)	9	0.41	1.00	1.00	1.00
TwoStage(8,1)	10	0.04	1.00	1.00	1.00
TwoStage(10,1)	12	0.00	1.00	1.00	1.00
Reorder(9,1)	11	0.06	1.00	1.00	1.00
Reorder(10,1)	12	0.00	1.00	1.00	1.00
Wronglock(1,20)	22	0.28	1.00	1.00	1.00
AbsList(1,7)	9	0.01	1.00	0.37	0.00
AbsList(1,8)	10	0.00	1.00	0.08	0.00
Deadlock(1,9)	11	0.00	1.00	1.00	1.00
Deadlock(1,10)	12	0.00	1.00	1.00	1.00
AryList(1,5)	7	0.81	1.00	1.00	1.00
AryList(1,8)	10	0.00	1.00	1.00	0.01
AryList(1,9)	11	0.00	1.00	1.00	0.00
AryList(1,10)	12	0.00	1.00	1.00	0.00

– **Deadlock:** Real, Num of locs: 6, Deadlock in the `Vector` and `Hashtable` classes due to a circular data dependency [12]. SLOC: 7151

3.2 Error Discovery

In Table 1 we compare the error densities of randomized depth-first search (`Random DFS`) to the meta heuristic using a greedy depth-first search. The error density which is a dependent variable in this study is defined as the probability of a technique finding an error in the program. To compute this probability we use the ratio of the number of error discovering trials over the total number of trials executed for a given model and technique. A technique that generates an error density of 1.00 is termed effective in error discovery while a technique that generates an error density of 0.00 is termed ineffective for error discovery.

We test three different secondary heuristics which is an independent variable to study the effect of the underlying heuristic on the effectiveness of the meta heuristic: (1) The polymorphic distance heuristic (`PFSM`) computes the distance between a target program location and the current program location on the control flow representation of the program. The heuristic rank based on the distance estimate lends itself naturally to guiding the search toward the next location in the sequence [11]. (2) The random heuristic (`Rand`) always returns a random value as the heuristic estimate. It serves as a baseline measure to test the effectiveness of guiding along the input sequence in the absence of any secondary guidance. (3) The prefer-thread heuristic (`Prefer Threads`) assigns a low heuristic value to a set of user-specified threads [8]. For example, if there are five total threads in a program then the user can specify to prefer the execution of certain threads over others when making scheduling choices.

Table 2. Comparison of the heuristics when used with the meta heuristic

Subject	PFSM Heuristic			Random Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
TwoStage(7,1)	209	213	217	40851	130839	409156	414187	2206109	4813016
TwoStage(8,1)	246	250	255	49682	217637	502762	609085	4436444	10025314
TwoStage(10,1)	329	333	340	52794	314590	827830	2635251	6690008	8771151
Wronglock(1,10)	804	3526	12542	73	7082	22418	560	120305	675987
Wronglock(1,20)	2445	21391	175708	67	24479	242418	1900	3827020	15112994
Reorder(5,1)	106	109	112	1803	5597	10408	259	977	2402
Reorder(8,1)	193	197	202	17474	36332	65733	523	3110	13536
Reorder(10,1)	266	271	277	28748	67958	110335	771	5136	16492
AryList(1,10)	1764	14044	55241	3652	15972	63206	-	-	-
AbsList(1,10)	1382	1382	1382	10497302	10497302	10497302	-	-	-

The results in Table 1 indicate that the meta heuristic, overall, has a higher error discovery rate compared to randomized depth-first search. In the `TwoStage` example the error density drops from 0.41 to 0.00 when going from the configuration of `TwoStage(7,1)` to the `TwoStage(10,1)` configuration. A similar pattern is observed in the `Reorder` model where the error density goes from 0.06 to 0.0; in the `AryList` model the error density drops from a respectable 0.81 to 0.00. For all these models, the meta heuristic using the polymorphic distance heuristic finds an error in every single trial as indicated by the error density of 1.00. In some cases, even when we use the random heuristic as the secondary heuristic, the greedy depth-first search outperforms the randomized depth-first search.

The `AbsList`, `AryList`, and `Deadlock` models represent real errors in the JDK 1.4 concurrent library. The `AbsList` model contains the portion of code shown in Fig. 1. In addition to the locations shown in Section 2.1 we manually add other data definition locations that are relevant in reaching the locations shown in Section 2.1. We use the meta heuristic to successfully generate a concrete error trace for the possible race condition reported by Jlint. The counter-example shows that the race-condition is caused because the `equals` method in Fig. 1(c) never acquires a lock on the input parameter. This missing lock allows another thread to modify the list (by adding an object on line 11 in Fig. 1(b)) while the thread is iterating over the list in the `equals` method. To our knowledge, this is the first report of the particular race condition in the JDK 1.4 library. It can be argued that the application using the library is incorrect and changing the comparison on line 13 of Fig. 1(b) to l_2.`equals`(l_1) can fix the error; however, we term it as a bug in the library because the usage of the library is in accordance with the documentation.

Table 2 reports the minimum, average, and maximum number of states generated in the error discovering trials of the meta heuristic using the three secondary heuristics. The entries in Table 2 marked "-" indicate that the technique was unable to find an error in 100 independent greedy depth-first search trials that are time-bounded at one hour. In the `TwoStage`, `Reorder`, `AryList`, `AbsList` subjects, the minimum, average, and maximum states generated by the PFSM heuristic is perceptibly less than the random and prefer-thread heuristics.

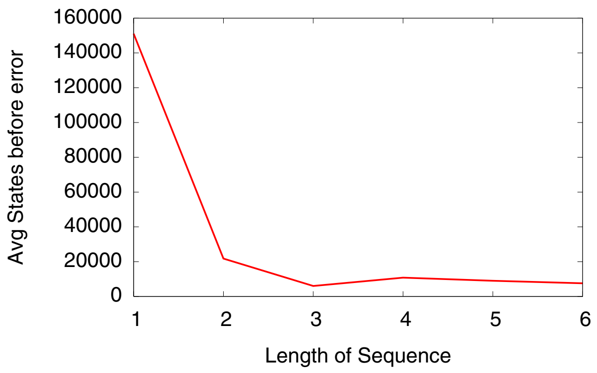


Fig. 6. Effect of varying the number of locations in the sequence in the AryLst(1,10) program to verify the race condition in the JDK1.4 concurrent library

Consider the `Twostage(7,1)` model where, on average, the PFSM heuristic only generates 213 states while the random heuristic and prefer-thread heuristic generate 130, 839 and 2, 206, 109 states respectively, on average, before error discovery. In the `AbsList(1,10)` model the PFSM heuristic finds the error every time by exploring a mere 1382 states. In contrast, from a total of 100 trials with the random heuristic only a single trial finds the error after exploring over a million states, while the prefer-thread heuristic is unable to find the error in the 100 trials. `Wronglock` is the only model where the minimum number of states generated by the random heuristic is less than the PFSM heuristic. This example shows that it is possible for the random heuristic to get just lucky in certain models. The results in Table 2 demonstrate that a better underlying secondary heuristic helps the meta heuristic generate fewer states before error discovery. The trends observed in Table 2 are also observed in total time taken before error discovery, total memory used, and length of counter-example.

3.3 Effect of the Sequence Length

We vary the number of key locations in the input sequence provided to the meta heuristic to study the effect of the number of locations on the performance of the meta heuristic. In Fig. 6 we plot the average number of states generated (the dependent variable) before error discovery while varying sequence lengths in the `AryLst` model. In Fig. 6 there is a sharp drop in the number of states when we increase the number of key locations from one to two. A smaller decrease in the average number of states is observed between sequence lengths two and three. We observe the effects of diminishing returns after three key locations and the number of states does not vary much. In general, for the models presented in this study, only 2–3 key locations are required for the meta heuristic to be effective. In the possible race condition shown in Fig. 1 (`AbsList` model), however, we needed to specify a minimum of five key program locations in the input sequence for the meta heuristic to find a corresponding concrete error trace. Recall that the `AbsList` model represents the race-condition in the `AbstractList` class while using the synchronized `Vector` sub-class in the JDK 1.4 library.

4 Related Work

Static analysis techniques ignore the actual execution environment of the program and reason about errors by simply analyzing the source code of the program. ESC/Java relies heavily on program annotations to find deadlocks and race-conditions in the programs [1]. Annotating existing code is cumbersome and time consuming. RacerX does a top-down inter-procedural analysis starting from the root of the program [2]. Similarly, the work by Williams *et al.* does a static deadlock detection in Java libraries [12]. FindBugs and Jlint look for suspicious patterns in Java programs [3,4]. Error warnings reported by static analysis tools have to be manually verified which is difficult and sometimes not possible. The output of such techniques, however, serve as ideal input for the meta heuristic presented in this paper. Furthermore, dynamic analysis techniques can also be used to generate warnings about potential errors in the programs [18,19].

Model checking is a formal approach for systematically exploring the behavior of a concurrent software system to verify whether the system satisfies the user specified properties [5,6]. In contrast to exhaustively searching the system, directed model checking uses heuristics to guide the search quickly toward the error [7,8,20,9,10,11]. Property-based heuristics and structural heuristics consider the property being verified and structure of the program respectively to compute a heuristic rank [7,8]. Distance estimate heuristics rank the states based on the distance to a possible error location [20,9,10,11]. As seen in the results, the PFSM distance heuristic is very effective in guiding the search toward a particular location; however, its success is dramatically improved in combination with the meta heuristic.

The trail directed model checking by Edelkamp *et. al* uses a concrete counter-example generated by a depth-first search as input to its guidance strategy [21]. It uses information from the original counter-example (trail) in order to generate an optimal counter-example. The goal in this work, however, is to achieve error discovery in models where exhaustive search techniques are unable to find an error. The deterministic execution technique used to test concurrent Java monitors is related to the technique presented in this paper [22]. The deterministic execution approach, however, requires a significant manual effort with the tester required to provide data values to execute different branch conditions, thread schedules, and sequence of methods.

Similar ideas of guiding the program execution using information from some abstraction of the system have been explored in hardware verification with considerable success [23,24]. An interesting avenue of future work would be to study the reasons for the success (in concretizing abstract traces by guiding program execution) that we observe in such disparate domains with very different abstraction and guidance strategies.

5 Conclusions and Future Work

This paper presents a meta heuristic that automatically verifies the presence of errors in real multi-threaded Java programs based on static analysis warnings. We provide the meta heuristic a sequence of locations and it automatically controls scheduling decisions to direct the execution of the program using a two-tier ranking scheme in a greedy-depth first manner. The study presented in this paper shows that the meta heuristic is effective in error discovery in subjects where randomized depth-first search fails to find an error. Using the meta heuristic we discovered real concurrency errors in the JDK 1.4 library. In future work we want to take the output of a static analysis tool and automatically generate the input sequence using control and data dependence analyses. Also we would like to extend the technique to handle non-determinism arising due to data values.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI, pp. 234–245. ACM, New York (2002)
2. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 237–252. ACM Press, New York (2003)
3. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. 39(12), 92–106 (2004)
4. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded java programs. In: Proc. ASWEC, Washington, DC, USA, p. 68. IEEE Computer Society, Los Alamitos (2001)
5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
6. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proc. ASE, Grenoble, France (September 2000)
7. Edelkamp, S., Lafuente, A.L., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, p. 57. Springer, Heidelberg (2001)
8. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: Proc. ISSTA, pp. 12–21 (2002)
9. Rungta, N., Mercer, E.G.: A context-sensitive structural heuristic for guided search model checking. In: Proc. ASE, Long Beach, California, USA, November 2005, pp. 410–413 (2005)
10. Rungta, N., Mercer, E.G.: An improved distance heuristic function for directed software model checking. In: Proc. FMCAD, Washington, DC, USA, pp. 60–67. IEEE Computer Society, Los Alamitos (2006)
11. Rungta, N., Mercer, E.G.: Guided model checking for programs with polymorphism. In: Proc. PEPM, Savannah, Georgia, USA (2009) (to appear)
12. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 602–629. Springer, Heidelberg (2005)

13. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS 2003: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, p. 286.2. IEEE Computer Society Press, Los Alamitos (2003)
14. Rungta, N., Mercer, E.G.: Generating counter-examples through randomized guided search. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 39–57. Springer, Heidelberg (2007)
15. Dwyer, M.B., Person, S., Elbaum, S.: Controlling factors in evaluating path-sensitive error detection techniques. In: Proc. FSE 2006, pp. 92–104. ACM Press, New York (2006)
16. Dwyer, M.B., Elbaum, S., Person, S., Purandare, R.: Parallel randomized state-space search. In: Proc. ICSE 2007, Washington, DC, USA, pp. 3–12. IEEE Computer Society, Los Alamitos (2007)
17. Rungta, N., Mercer, E.G.: Hardness for explicit state software model checking benchmarks. In: Proc. SEFM 2007, London, U.K, September 2007, pp. 247–256 (2007)
18. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: Proc. SPIN Workshop, London, UK, pp. 245–264. Springer, Heidelberg (2000)
19. Shacham, O., Sagiv, M., Schuster, A.: Scaling model checking of dataraces using dynamic information. J. Parallel Distrib. Comput. 67(5), 536–550 (2007)
20. Edelkamp, S., Mehler, T.: Byte code distance heuristics and trail direction for model checking Java programs. In: Proc. MoChArt, pp. 69–76 (2003)
21. Edelkamp, S., Lafuente, A.L., Leue, S.: Trail-directed model checking. In: Stoller, S.D., Visser, W. (eds.) ENTCS, vol. 55. Elsevier Science Publishers, Amsterdam (2001)
22. Harvey, C., Strooper, P.: Testing Java monitors through deterministic execution. In: Proc. ASWEC, Washington, DC, USA, p. 61. IEEE Computer Society, Los Alamitos (2001)
23. Nanshi, K., Somenzi, F.: Guiding simulation with increasingly refined abstract traces. In: Proc. DAC, pp. 737–742. ACM, New York (2006)
24. Paula, F.M.D., Hu, A.J.: An effective guidance strategy for abstraction-guided simulation. In: Proc. DAC 2007, pp. 63–68. ACM, New York (2007)

A Uniform Approach to Three-Valued Semantics for μ-Calculus on Abstractions of Hybrid Automata

K. Bauer, R. Gentilini, and K. Schneider

Department of Computer Science, University of Kaiserslautern, Germany
{k_bauer,gentilin,schneider}@informatik.uni-kl.de

Abstract. Abstraction/refinement methods play a central role in the analysis of hybrid automata, that are rarely decidable. Soundness (of evaluated properties) is a major challenge for these methods, since abstractions can introduce unrealistic behaviors.

In this paper, we consider the definition of a three-valued semantics for μ-calculus on abstractions of hybrid automata. Our approach relies on two steps: First, we develop a framework that is general in the sense that it provides a preservation result that holds for several possible semantics of the modal operators. In a second step, we instantiate our framework to two particular abstractions. To this end, a key issue is the consideration of both over- *and* under-approximated reachability analysis, while classic simulation-based abstractions rely only on overapproximations, and limit the preservation to the universal (μ-calculus') fragment. To specialize our general result, we consider (1) so-called discrete bounded bisimulation abstractions, and (2) modal abstractions based on may/must transitions.

1 Introduction

Hybrid automata [16,1] provide an appropriate modeling paradigm for systems where continuous variables interact with discrete modes. Such models are frequently used in complex engineering fields like embedded systems, robotics, avionics, and aeronautics [2,12,24,25]. In hybrid automata, the interaction between discrete and continuous dynamics is naturally expressed by associating a set of differential equations to every location of a finite automaton.

Finite automata and differential equations are well established formalisms in mathematics and computer science. Despite of their long-standing tradition, their combination in form of hybrid automata leads to surprisingly difficult problems that are often undecidable. In particular, the *reachability* problem is undecidable for most families of hybrid automata [1,14,20,21,22], and the few decidability results are built upon strong restrictions of the dynamics [3,17]. The reachability analysis of hybrid automata is a fundamental task, since checking *safety* properties of the underlying system can be reduced to a reachability problem for the set of bad configurations [16].

For this reason, a growing body of research is being developed on the issue of dealing with approximated reachability on undecidable – yet reasonably expressive – hybrid automata [9,11,23,25,26]. To this end, most of the techniques proposed so far either rely on bounded state-reachability or on the definition of finite abstractions. While the first approach suffers inherently of incompleteness, the quest for *soundness* is a key issue in the context of methods based on abstractions. In fact, abstractions can introduce unrealistic behaviors that may yield to spurious errors being reported in the safety analysis. Usually, a simulation preorder is required to relate the abstraction to the concrete dynamics of the hybrid system under consideration, ensuring at least the correctness of each response of (abstract) *non* reachability.

In this work, we provide a *uniform* approach to the sound evaluation of *general* reactive properties on abstractions of hybrid automata. Here, 'general' refers to the fact that we specify properties by means of the highly expressive logic of μ-calculus, that covers in particular CTL and other specification logics. 'Uniform', instead, emphasizes that we consider different possible classes of abstractions, whose analysis permits to recover both under- and overapproximations of state-sets fulfilling a given reachability requirement. Intuitively, this requirement is a minimal prerequisite for recovering sound abstract evaluations of arbitrary μ-calculus formulas.

To achieve our results we proceed by two steps: We start with the development of a generic semantics scheme for the μ-calculus, where the meaning of the modal operators can be adapted to particular abstractions. Assuming certain constraints for the semantics of these operators, we can prove a preservation result for our generic semantics scheme, thus providing a general framework for different classes of abstractions. In a subsequent step, we specialize our framework to suitable abstractions. In particular, we demonstrate the applicability of our framework by considering (1) the class of so-called discrete bounded bisimulation (DBB) abstractions [10], and (2) a kind of *modal* abstractions based on may/must transitions. As a final contribution, we compare these instances of our framework with respect to the issue of *monotonicity* of preserved μ-calculus formulas.

The paper is organized as follows: Preliminaries are given in Section 2. Section 3 introduces the classes of abstractions used in Section 5 to instantiate the generic result on preservative three-valued μ-calculus semantics outlined in Section 4. The monotonicity issue is dealt with in Section 6, while Section 7 concludes the paper discussing its contributions. All proofs are given in the appendix and in [4].

2 Preliminaries

In this section, we introduce the basic notions used in the remainder of the paper.

Definition 1 (Hybrid Automata [3]). *A* Hybrid Automaton *is a tuple* $H = \langle L, E, X, \mathit{Init}, \mathit{Inv}, F, G, R\rangle$ *with the following components:*

- *a finite set of* locations L
- *a finite set of* discrete transitions *(or* jumps*)* $E \subseteq L \times L$
- *a finite set of* continuous variables $X = \{x_1, \dots x_n\}$ *that take values in* $\mathbb{R}$
- *an initial set of conditions:* $\mathit{Init} \subseteq L \times \mathbb{R}^n$
- $\mathit{Inv} : L \to 2^{\mathbb{R}^n}$*, the* invariant location labeling
- $F : L \times \mathbb{R}^n \to \mathbb{R}^n$*, assigning to each location* $\ell \in L$ *a vector field* $F(\ell, \cdot)$ *that defines the evolution of continuous variables within* ℓ
- $G : E \to 2^{\mathbb{R}^n}$*, the* guard edge labeling
- $R : E \times \mathbb{R}^n \to 2^{\mathbb{R}^n}$*, the* reset edge labeling.

We write $\mathbf{v}$ to represent a valuation $(v_1, \dots, v_n) \in \mathbb{R}^n$ of the variables' vector $\mathbf{x} = (x_1, \dots, x_n)$, whereas $\dot{\mathbf{x}}$ denotes the first derivatives of the variables in $\mathbf{x}$ (they all depend on the time, and are therefore rather functions than variables). A *state* in H is a pair $s = (\ell, \mathbf{v})$, where $\ell \in L$ is called the *discrete component* of s and $\mathbf{v}$ is called the *continuous component* of s. A *run* of H is a path in the *time abstract transition system* of H, given in Definition 2.

Definition 2. *The* time abstract transition system *of the hybrid automaton $H = \langle L, E, X, Init, Inv, F, G, R\rangle$ is the transition system $T_H = \langle Q, Q_0, \ell_{\rightarrow}, \rightarrow\rangle$, where:*

- $Q \subseteq L \times \mathbb{R}^n$ *and* $(\ell, \mathbf{v}) \in Q$ *if and only if* $\mathbf{v} \in Inv(\ell)$
- $Q_0 \subseteq Q$ *and* $(\ell, \mathbf{v}) \in Q_0$ *if and only if* $\mathbf{v} \in Init(\ell) \cap Inv(\ell)$
- $\ell_{\rightarrow} = \{e, \delta\}$ *is the set of edge labels, that are determined as follows:*
  - *there is a* continuous transition $(\ell, \mathbf{v}) \xrightarrow{\delta} (\ell, \mathbf{v}')$, *if and only if there is a differentiable function* $f : [0, t] \rightarrow \mathbb{R}^n$, *with* $\dot{f} : [0, t] \rightarrow \mathbb{R}^n$ *such that:*
    1. $f(0) = \mathbf{v}$ *and* $f(t) = \mathbf{v}'$
    2. *for all* $\varepsilon \in (0, t)$, $f(\varepsilon) \in Inv(\ell)$, *and* $\dot{f}(\varepsilon) = F(\ell, f(\varepsilon))$.
  - *there is a* discrete transition $(\ell, \mathbf{v}) \xrightarrow{e} (\ell', \mathbf{v}')$ *if and only if there exists an edge* $(\ell, \ell') \in E$, $\mathbf{v} \in G(\ell)$ *and* $\mathbf{v}' \in R((\ell, \ell'), \mathbf{v})$.

Definition 3 and Definition 4 recapitulate the syntax and the semantics of the μ-calculus language L_μ on hybrid automata, respectively [6,7].

Definition 3 (L_μ Syntax). *The set of μ-calculus preformulas for a set of labels $a \in \{e, \delta\}$ and propositions $p \in AP$ is defined by the following syntax:*

$$\phi := \ p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a\rangle\phi \mid [a]\phi \mid E(\phi_1 \underline{U} \phi_2) \mid A(\phi_1 \underline{U} \phi_2) \mid \mu Z.\phi \mid \nu Z.\phi$$

The set L_μ of μ-calculus formulas is defined as the subset of pre-formulas, where each subformula of the type $\mu Z.\phi$ and $\nu Z.\phi$ satisfies that all occurrences of Z in ϕ occur under an even number of negation symbols.

Definition 4 (Semantics of L_μ on Hybrid Automata). *Let AP be a finite set of propositional letters, let $p \in AP$ and consider $H = \langle L, E, X, Init, Inv, F, G, R\rangle$. Given $\ell_{\mathsf{AP}} : Q \rightarrow 2^{\mathsf{AP}}$ and $\phi \in L_\mu$, the function $[\![\phi]\!] : Q \rightarrow \{0, 1\}$ is inductively defined:*

- $[\![p]\!](q) = 1$ *iff* $p \in l_{AP}(q)$
- $[\![\neg\phi]\!] := \neg\, [\![\phi]\!]$
- $[\![\phi \diamond \psi]\!] := [\![\phi]\!] \diamond [\![\psi]\!]$ *for* $\diamond \in \{\vee, \wedge\}$
- $[\![E(\phi \underline{U} \psi)]\!](q) = 1$ *iff there exists a run ρ departing from q that admits a prefix* $\rho^* := q_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} q_n$, *where* $q = q_1$, $a_i \in \{e, \delta\}$, $q_i = (l, v_i)$, *satisfying:*
  · $[\![\psi]\!](q_n) = 1$ *and for* $1 \le i < n$: $[\![\phi]\!](q_i) = 1$
  · *for* $a_i = \delta$: $\exists$ *a differentiable function* $f : [0, t] \rightarrow \mathbb{R}^n$, *for which:*
    1. $f(0) = \mathbf{v_i}$ *and* $f(t) = \mathbf{v_{i+1}}$
    2. *for all* $\varepsilon \in (0, t)$, $f(\varepsilon) \in Inv(\ell)$, *and* $\dot{f}(\varepsilon) = F(\ell, f(\varepsilon))$
    3. *for all* $\varepsilon \in (0, t)$, $q' = (l_i, f(\varepsilon))$ *satisfies* $[\![\phi \vee \psi]\!](q') = 1$
- $[\![A(\phi \underline{U} \psi)]\!](q_1) = 1$ *iff for all runs ρ departing from q there exists a prefix* $\rho^* := q_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} q_n$, *where* $q = q_1$, $a_i \in \{e, \delta\}$, $q_i = (l, v_i)$, *satisfying:*

  · $[\![\psi]\!](q_n) = 1$ *and for* $1 \leq i < n$: $[\![\phi]\!](q_i) = 1$
  · *for* $a_i = \delta$: $\exists$ *a differentiable function* $f : [0,t] \to \mathbb{R}^n$, *for which:*
    1. $f(0) = \mathbf{v_i}$ *and* $f(t) = \mathbf{v_{i+1}}$
    2. *for all* $\varepsilon \in (0,t)$, $f(\varepsilon) \in Inv(\ell)$, *and* $\dot{f}(\varepsilon) = F(\ell, f(\varepsilon))$
    3. *for all* $\varepsilon \in (0,t)$, $q' = (l_i, f(\varepsilon))$ *satisfies* $[\![\phi \vee \psi]\!](q') = 1$

- $[\![\langle a \rangle \phi]\!](q) = 1$ *iff* $\exists q \xrightarrow{a} q' : [\![\phi]\!](q') = 1$
  $[\![[a]\phi]\!](q) = 1$ *iff* $\forall q \xrightarrow{a} q' : [\![\phi]\!](q') = 1$
- *The fixpoint operators are defined in the following way:*
  Let $[\phi]_Z^\psi$ *be the formula obtained by replacing all occurrences of* Z *with* ψ. *Given a fixpoint formula* $\sigma Z.\phi$ *with* $\sigma \in \{\mu, \nu\}$ *its k-th approximation* $apx_k(\sigma Z.\phi)$ *is recursively defined as follows:*

$$apx_0(\mu Z.\phi) := 0 \text{ and } apx_{k+1}(\mu Z.\phi) := [\phi]_Z^{apx_k(\mu Z.\phi)}$$
$$apx_0(\nu Z.\phi) := 1 \text{ and } apx_{k+1}(\nu Z.\phi) := [\phi]_Z^{apx_k(\nu Z.\phi)}$$

  Then smallest and greatest fixpoints $[\![\sigma Z.\phi]\!]$ *are defined by*
  · *smallest fixpoint:* $[\![\mu Z.\phi]\!] := \bigvee_{k \in \mathbb{N}} [\![apx_k(\mu Z.\phi)]\!]$
  · *greatest fixpoint:* $[\![\nu Z.\phi]\!] := \bigwedge_{k \in \mathbb{N}} [\![apx_k(\mu Z.\phi)]\!]$

  $H \models \phi$ *iff* $\forall q_0 \in Q_0 : [\![\phi]\!](q_0) = 1$.

The following definition recalls the notion of *simulation* relation, that plays a central role in the context of hybrid automata abstractions.

Definition 5 (Simulation). *Let* $T_1 = \langle Q^1, Q_0^1, \ell_{\to}, \to^1 \rangle$, $T_2 = \langle Q^2, Q_0^2, \ell_{\to}, \to^2 \rangle$, $Q^1 \cap Q^2 = \emptyset$, *be two edge-labeled transition systems and let* $\mathcal{P}$ *be a partition on* $Q_1 \cup Q_2$. *A* simulation *from* T_1 *to* T_2 *is a non-empty relation on* $\rho \subseteq Q^1 \times Q^2$ *such that, for all* $(p,q) \in \rho$:

- $p \in Q_0^1$ *iff* $q \in Q_0^2$ *and* $[p]_{\mathcal{P}} = [q]_{\mathcal{P}}$.
- *for each label* $a \in \ell_{\to}$, *if there exists* p' *such that* $p \xrightarrow{a} p'$, *then there exists* q' *such that* $(p', q') \in \rho$ *and* $q \xrightarrow{a} q'$.

If there exists a simulation from T_1 *to* T_2, *then we say that* T_2 simulates T_1, *denoted* $T_1 \leq_S T_2$. *If* $T_1 \leq_S T_2$ *and* $T_2 \leq_S T_1$, *then* T_1 *and* T_2 *are said* similar, *denoted* $T_1 \equiv_S T_2$. *If* ρ *is a simulation from* T_1 *to* T_2, *and the inverse relation* ρ^{-1} *is a simulation from* T_2 *to* T_1, *then* T_1 *and* T_2 *are said* bisimilar, *denoted* $T_1 \equiv_B T_2$

3 Abstractions of Hybrid Automata for Parallel over- and Underapproximated Reachability Analysis

In this section, we introduce two kinds of abstractions that we will use in the sequel to specialize our general preservation result for μ-calculus semantics.

Most of the abstraction/refinement methods for hybrid automata in the literature are based on overapproximations of the reachable states[1]. In particular, they rely on a generic notion of abstractions based on the simulation preorder. The latter is required to relate the abstraction to the dynamics of the hybrid automaton, as formalized below.

[1] Note that the reachability problem is undecidable for most classes of hybrid automata.

Definition 6 (Abstraction). *Let H be a hybrid automaton. An abstraction of H is a finite transition system $A = \langle R, R_0, \xrightarrow{\delta}, \xrightarrow{e} \rangle$ in which*

1. *R is a finite partition of the state space of H, $R_0 \subseteq R$ is a partition of the initial states, $\xrightarrow{\delta} \subseteq R \times R$ and $\xrightarrow{e} \subseteq R \times R$*
2. *$A^* := \langle R, R_0 \overset{\delta}{\rightarrow}^*, \xrightarrow{e} \rangle$ simulates the time abstract transition system T_H associated to H, where $\overset{\delta}{\rightarrow}^*$ denotes the transitive closure of the continuous transitions $\xrightarrow{\delta}$*

Since this basic notion of abstraction gives only an overapproximation of the hybrid automaton's reachable states, its usage is inherently limited to the universal fragment of the μ-calculus [5]. As we are interested in unrestricted μ-calculus properties, we need a more powerful abstraction/refinement approach. To this end, a minimum requirement is the combination of both over- and underapproximations of state-sets satisfying a given reachability property. The consideration of parallel over- and underapproximated reachability on hybrid automata is quite new: In [10], discrete bounded bisimulation (DBB) abstractions, briefly recalled in Subsection 3.1, were designed for this purpose. Another approach that leads to over- and underapproximations is given by *modal abstractions* for hybrid automata, that we develop in Subsection 3.2 (generalizing the definitions given in context of discrete systems [13]).

3.1 Discrete Bounded Bisimulation (DBB) Abstractions

It is well known that the classic bisimulation equivalence can be characterized as a coarsest partition stable with respect to a given transition relation [18]. Bounded bisimulation imposes a bound on the number of times each edge can be used for partition refinement purposes. For the equivalence of discrete bounded bisimulation (DBB), the latter bound applies only to the *discrete* transitions of a given hybrid automaton, as recalled in Definition 7, below.

Definition 7 (Discrete Bounded Bisimulation [10]). *Let H be an hybrid automaton, and consider the partition $\mathcal{P}$ on the state-space Q of $T_H = \langle Q, Q_0, \ell_{\rightarrow}, \rightarrow \rangle$. Then:*

1. *$\equiv_0 \in Q \times Q$ is the maximum relation on Q such that for all $p \equiv_0 q$*
   (a) $[p]_P = [q]_P$ and $p \in Q_0$ iff $q \in Q_0$
   (b) $\forall p \xrightarrow{\delta} p' \exists q' : p' \equiv_0 q' \wedge q \xrightarrow{\delta} q'$
   (c) $\forall q \xrightarrow{\delta} p' \exists q' : p' \equiv_0 q' \wedge p \xrightarrow{\delta} p'$
2. *$\equiv_n \in Q \times Q$ is the maximum relation on Q such that for all $p \equiv_n q$*
   (a) $p \equiv_{n-1} q$
   (b) $\forall p \xrightarrow{\delta} p' \exists q' : p' \equiv_n q' \wedge q \xrightarrow{\delta} q'$
   (c) $\forall q \xrightarrow{\delta} p' \exists p' : p' \equiv_n q' \wedge p \xrightarrow{\delta} p'$
   (d) $\forall p \xrightarrow{e} p' \exists q' : p' \equiv_{n-1} q' \wedge q \xrightarrow{e} q'$
   (e) $\forall q \xrightarrow{e} q' \exists p' : p' \equiv_{n-1} q' \wedge p \xrightarrow{e} p'$

For $n \in \mathbb{N}$, the relation $\equiv_n$ will be called n-DBB equivalence.

The succession of n-DBB equivalences over an hybrid automaton H naturally induces a series of abstractions for H, as stated in Definition 8.

Definition 8 (Series of DBB Abstractions [10]). *Let H be a hybrid automaton and $T_H = \langle Q, Q_0, l_{\rightarrow}, \rightarrow\rangle$ be the associated time abstract transition system. Let $\mathcal{P}$ be a partition of Q and consider the n-DBB equivalence $\equiv_n$. Then, the n-DBB abstraction $H_{\equiv_n} = \langle Q', Q'_0, l_{\rightarrow} \rightarrow'\rangle$ is defined as follows:*

– $Q' = Q_{/\equiv_n}$, $Q'_0 = Q_{0/\equiv_n}$
– $\forall \alpha, \beta \in Q'$:
  • $\alpha \xrightarrow{e} \beta$ *iff* $\exists a \in \alpha \exists b \in \beta : a \xrightarrow{e} b$
  • $\alpha \xrightarrow{\delta} \beta$ *iff* $\exists a \in \alpha \exists b \in \beta : a \xrightarrow{\delta} b$ *and the path* $a \rightsquigarrow b$ *only traverses* α *and* β

The existence of a simulation preorder relating successive elements in a series of DBB abstractions allows the refinement of overapproximations of reachable sets in the considered hybrid automaton [10]. Moreover, $H_{\equiv_n}$ preserves the reachability of a given region of interest (in the initial partition) whenever the latter can be established on H following a path that traverses at most n locations [10]. On this ground, it is also possible to use the succession of DBB abstractions to obtain $\subseteq$-monotonic underapproximations of the set of states fulfilling a given reachability requirement.

3.2 Modal Abstractions Based on May/Must Relations

For discrete systems [13], a *may*-transition between two abstract classes r and r' encodes that for at least some state in r there is a transition to some state in r'. In turn, a *must*-transition between r and r' states that all states in r have a transition to a state in r'. Naturally, may-transitions (resp. must-transitions) refer to overapproximated (resp. underapproximated) transitions among classes of an abstract system. The above ideas can be extended to the context of hybrid automata as formalized in Definition 9.

Definition 9 (Modal Abstractions). *Let $A = \langle R, R_0, \xrightarrow{\delta}, \xrightarrow{e}\rangle$ be an abstraction of the hybrid automaton H. Then A is a* modal abstraction *(or* may/must abstraction*) of H iff the following properties hold:*

– $\xrightarrow{\delta} \supseteq \xrightarrow{\delta}_{must}$, *where* $\xrightarrow{\delta}_{must}$ *is defined as follows:*
  $r \xrightarrow{\delta}_{must} r'$ *iff for all $x \in r$ there exists an $x' \in r'$ such that H can evolve continuously from the state x to the state x' by traversing the only regions* r *and* r'.
– $\xrightarrow{e} \supseteq \xrightarrow{e}_{must}$ *where* $\xrightarrow{e}_{must}$ *is defined as follows:*
  $r \xrightarrow{e}_{must} r'$ *iff for all $x \in r$ there exists an $x' \in r'$ s.t. $x \xrightarrow{e} x'$ in H.*

The subautomaton $\langle R, R_0, \xrightarrow{\delta}_{must}, \xrightarrow{e}_{must}\rangle$ of A is called A_{must}.

Given the modal abstraction A for the hybrid automaton H, Lemma 1 states that A_{must} is simulated by the time abstract transition system T_H of H.

Lemma 1. *Let H be a hybrid automaton and let A be a may/must abstraction of H. Then, the subautomaton A_{must} of A is simulated by T_H, i.e. $A_{must} \leq_S T_H \leq_S A^*$.*

On this ground, may/must abstractions can be used not only to overapproximate, but also to underapproximate the set of states modeling a given reachability property, as stated in Corollary 1.

Corollary 1. *Let $A = \langle R, R_0 \overset{\delta}{\rightarrow}, \overset{e}{\rightarrow}\rangle$ be a modal abstraction for the hybrid automaton H, and let F be a set of (final) states in H. Assume that F is consistent with respect to R, i.e. for all $r \in R : r \cap F = r \ \vee \ r \cap F = \emptyset$. If $r \in R$ admits a path to $r' \subseteq F$ in $\mathcal{A}_{must}$, then for all $s \in r$, exists $s' \in r'$ such that H admits a run from s to s'.*

4 A Generic Semantics for μ-Calculus on Abstractions of Hybrid Automata

In this section, we present one of the main ingredients of our approach: a *generic three-valued* semantics for μ-calculus on abstractions of hybrid automata. Here, two key-words deserve our attention: Generic and three-valued.

The choice of a three-valued logic as the base of our semantics is motivated by the broad family of abstractions that we consider for our framework. In fact, the abstractions we have in mind are in general less precise than a bisimulation (which allows for exact reachability analysis, but is seldom finite), and more precise than a simulation (that allows only for overapproximated reachability analysis). Hence, we can not expect that *any* μ-calculus formula is preserved, however it should be possible to recover at least all true universal μ-calculus subformulas[2]. By means of a three-valued logic, we can use the third value $\perp$ to distinguish the cases for which it is not possible to derive a boolean truth value, due to the coarseness of the abstraction. Instead, the preservation applies to all the boolean results established using the abstract semantics. In the following, we write $\neg_3$, $\vee_3$, $\wedge_3$ for the three-valued extensions of the boolean operations $\neg$, $\vee$, $\wedge$, respectively[3].

The second keyword – generic – is better understood as a way of establishing a link between (1) the quest for soundness in our semantics, and (2) the variety of patterns according to which different abstractions split the information over their regions. Our generic semantics is an abstract semantics scheme, where the evaluation is fixed for some operators (namely boolean and fixpoint operators), and only subject to some constraints for the others. The constraints are sufficient to establish a general preservation result, though the semantics scheme can be adapted to several classes of abstractions.

Given the above premises, we are now ready to formalize in Definition 10 our three-valued generic semantics for μ-calculus on abstractions of hybrid automata. Note that for a μ-calculus formula ϕ, we distinguish between the semantics $[\![\phi]\!]_H$ on a hybrid automaton H (as given in Definition 4) and the semantics $[\![\phi]\!](r)$ on the region r of an abstraction of H.

Definition 10 (Generic μ-Calculus Semantics). *Let H be a hybrid automaton whose state space is partitioned into finitely many regions of interest by the labeling function $l_{AP} : Q \rightarrow 2^{AP}$, where AP is a finite set of propositional letters. Let ϕ be a*

[2] Recall that bisimulation preserves the whole μ-calculus, while simulation preserves the only true universally quantified formulas.

[3] We use Kleene's definition of three-valued logic [19].

μ-calculus formula with atomic propositions AP, and consider the abstraction $A = \langle R, R_0, l_{\rightarrow}, \rightarrow\rangle$ where R is assumed to refine[4] *the regions of interest in H.*

1. *If ϕ is an atomic proposition, then* $[\![\phi]\!](r) = \begin{cases} 1 & \phi \in l_{AP}(r) \\ 0 & \textit{otherwise} \end{cases}$
2. *If $\phi = \neg\varphi$, then $[\![\neg\varphi]\!] = \neg_3[\![\varphi]\!]$*
3. *If $\phi = \varphi \vee \psi$, then $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \vee_3 [\![\psi]\!]$*
4. *If $\phi = \varphi \wedge \psi$, then $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \wedge_3 [\![\psi]\!]$*
5. *If $\phi \in \{\langle\delta\rangle\varphi, \langle e\rangle\varphi, [\delta]\varphi, [e]\varphi, E(\varphi \underline{U} \psi), A(\varphi \underline{U} \psi)\}$, then $[\![\phi]\!]$ is required to fulfill the following conditions:*

$$[\![\phi]\!](r) = 1 \Rightarrow \ \forall\, x \in r : \ [\![\phi]\!]_H(x) = 1$$
$$[\![\phi]\!](r) = 0 \Rightarrow \ \forall\, x \in r : \ [\![\phi]\!]_H(x) = 0$$

6. *Let $\phi \in \{\mu Z.\varphi, \nu Z.\varphi\}$ be a fixpoint formula. Let $[\varphi]^{\psi}_{Z}$ be the formula obtained by replacing all occurrences of Z with ψ. Given a fixpoint formula $\sigma Z.\varphi$ with $\sigma \in \{\mu, \nu\}$, its k-th approximation $apx_k(\sigma Z.\varphi)$ is recursively defined as follows:*
   - *$apx_0(\mu Z.\varphi) := 0$ and $apx_{k+1}(\mu Z.\varphi) := [\varphi]^{apx_k(\mu Z.\varphi)}_{Z}$*
   - *$apx_0(\nu Z.\varphi) := 1$ and $apx_{k+1}(\nu Z.\varphi) := [\varphi]^{apx_k(\nu Z.\varphi)}_{Z}$*

   The semantics of least and greatest fixpoints $[\![\sigma Z.\varphi]\!]$ are defined by $[\![apx_{\hat{k}}\sigma Z.\varphi]\!]$ where $\hat{k}$ is the smallest index where $[\![apx_{\hat{k}}(\sigma Z.\varphi)]\!] = [\![apx_{\hat{k}+1}(\sigma Z.\varphi)]\!]$ holds.

Let ϕ be a μ-calculus formula and let $A = \langle R, R_0 \xrightarrow{\delta}, \xrightarrow{e}\rangle$ be an abstraction of the hybrid automaton H. On the ground of Definition 10, we can define a three-valued relation $\models_3$ stating whether A is a model of the formula ϕ:

$$A \models_3 \phi = \begin{cases} 1 & \forall r \in R_0 : [\![\phi]\!](r) = 1 \\ 0 & \exists r \in R_0 : [\![\phi]\!](r) = 0 \\ \bot & otherwise \end{cases}$$

Theorem 1 below states that both results true and false established on A via $\models_3$ are preserved on the underlying hybrid automaton. Note that Theorem 1 has a sort of *uniform* character, since $\models_3$ subsumes indeed many possible effective semantics for μ-calculus, the latter recovered by specializing the semantics of the modal operators according to the properties of different classes of abstractions. For the rest of this work let $\preceq$ be the partial order over $\{0, 1, \bot\}$ defined by the reflexive closure of $\{(\bot, 0), (\bot, 1)\}$.

Theorem 1 (Uniform Preservation Theorem). *Let H be a hybrid automaton, let A be an abstraction of H. Then, for any μ-calculus formula ϕ, we have $A \models_3 \phi \preceq H \models \phi$.*

Hence, if $A \models_3 \phi$ is 1, so is $H \models \phi$, and if $A \models_3 \phi$ is 0, so is $H \not\models \phi$, and if $A \models_3 \phi$ is $\bot$, then $H \models \phi$ is completely unknown. For this reason, both valid and invalid subformulas can be preserved with our framework as long as the abstraction is not too coarse.

[4] Note that our assumption (the partition of Q into regions of interest is refined by the abstraction $A = \langle R, R_0, l_{\rightarrow}, \rightarrow\rangle$) implies that $\forall r \in R\, \forall x_1, x_2 \in R : \ l_{AP}(x_1) = l_{AP}(x_2)$ holds. Thus, the labeling function can be easily extended to $l_{AP} : R \rightarrow 2^{AP}$.

5 Instantiation to DBB- and May/Must-Abstractions

In this section, we specialize the general preservation result given in Section 4 to two particular instances, namely to modal and DBB abstractions. As a result, we obtain two preservative abstraction/refinement frameworks for μ-calculus on hybrid automata.

5.1 Semantics Completion for May/Must-Abstractions

In modal abstractions, each $\xrightarrow{\delta}_{must}$ (resp. $\xrightarrow{e}_{must}$) edge underapproximates a continuous (resp. discrete) evolution for the underlying hybrid automata. In turn, each $\xrightarrow{\delta}$ (resp. $\xrightarrow{e}$) edge overapproximates a continuous (resp. discrete) evolution for the considered hybrid automaton. The above considerations can be used to properly instantiate the semantics for the modal operators on may/must abstractions, completing the semantics scheme given in Definition 10. Consider for example the modal operator $\langle\delta\rangle$. According to the adaptive semantics scheme in Definition 10, we should instantiate the semantics $[\![\langle\delta\rangle\varphi]\!]$ in such a way that whenever $[\![\langle\delta\rangle\varphi]\!]$ evaluates to 1 (resp. 0) on an abstract region, then it evaluates to 1 (resp. 0) on all the states of the region. This constraint is naturally guaranteed on modal abstractions if we use only $\xrightarrow{\delta}_{must}$ edges (resp. $\xrightarrow{\delta}$ edges) to inspect for true (resp. false) evaluations. A similar way of reasoning allows to completely adapt the semantics scheme in Definition 10 to the case of modal abstractions, as formalized in Definition 11.

Definition 11. *Let H be a hybrid automaton, $A = \langle R, R_0 \xrightarrow{\delta}, \xrightarrow{e}\rangle$ be a may/must abstraction of H and let φ and ψ be μ-calculus formulas. Then, the semantics of the three-valued μ-calculus of Definition 10 for $a, a_i \in \{\delta, e\}$ is completed by:*

- $[\![\langle e\rangle\varphi]\!](r) \begin{cases} 1 & \exists r \xrightarrow{e}_{must} r' : [\![\varphi]\!](r') = 1 \\ 0 & \forall r \xrightarrow{e} r' : \quad [\![\varphi]\!](r') = 0 \\ \bot & \textit{otherwise} \end{cases}$
- $[\![\langle \delta\rangle\varphi]\!](r) \begin{cases} 1 & \exists r \xrightarrow{\delta}_{must} r' : [\![\varphi]\!](r') = 1 \\ 0 & \forall r \xrightarrow{\delta}{}^{*} r' : \quad [\![\varphi]\!](r') = 0 \\ \bot & \textit{otherwise} \end{cases}$
- $[\![[a]\phi]\!] = [\![\neg(\langle a\rangle\neg\phi)]\!]$
- *Let $\{r_n\}_{n\in\mathbb{N}}$ (resp. $\{r_n\}^{must}_{n\in\mathbb{N}}$) denote an infinite path of A (resp. A_{must}) starting in $r = r_0$. Then:*

$$[\![E(\varphi\underline{U}\psi)]\!](r) \begin{cases} 1 & \exists\{r_n\}^{must}_{n\in\mathbb{N}}\exists k \in \mathbb{N} : [\![\psi]\!](r_k) = 1 \wedge [\![\varphi]\!](r_{i<k}) = 1 \\ 0 & \forall\{r_n\}_{n\in\mathbb{N}}\forall k \in \mathbb{N} : \; [\![\psi]\!](r_k) \neq 0 \Rightarrow \exists i < k : [\![\varphi]\!](r_i) = 0 \\ \bot & \textit{otherwise} \end{cases}$$

$$[\![A(\varphi\underline{U}\psi)]\!](r) \begin{cases} 1 & \forall\{r_n\}_{n\in\mathbb{N}}\exists k \in \mathbb{N} : \; [\![\psi]\!](r_k) = 1 \; \wedge \; [\![\varphi]\!](r_{i<k}) = 1 \\ 0 & \exists\{r_n\}^{must}_{n\in\mathbb{N}}\forall k \in \mathbb{N} : [\![\psi]\!](r_k) \neq 0 \Rightarrow \exists i < k : [\![\varphi]\!](r_i) = 0 \\ \bot & \textit{otherwise} \end{cases}$$

Lemma 2, below, states the correctness of our instantiation, namely it ensures that the semantics for the modal operators on may/must abstractions in Definition 11 fulfill the constraints provided in Definition 10.

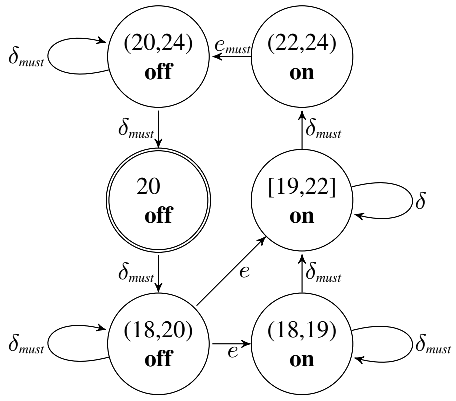


Fig. 1. May/Must Abstraction A_1 of the Heating Controller

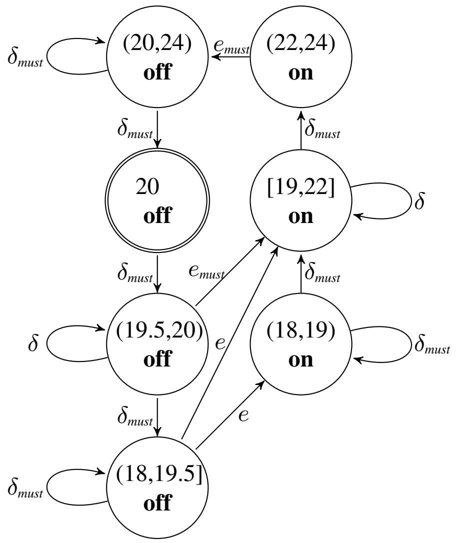


Fig. 2. May/Must Abstraction A_2 of the Heating Controller

Lemma 2. *Let A be a modal abstraction for the hybrid automaton H, and assume to interpret μ-calculus formulas according to Definition 11. Then, for any formula $\phi \in \{\langle\delta\rangle\varphi, \langle e\rangle\varphi, [\delta]\varphi, [e]\varphi, E(\varphi\underline{U}\psi), A(\varphi\underline{U}\psi)\}$, we have:*

$$\llbracket\phi\rrbracket(r) = 1 \Rightarrow \forall\, x \in r : \llbracket\phi\rrbracket_H(x) = 1$$
$$\llbracket\phi\rrbracket(r) = 0 \Rightarrow \forall\, x \in r : \llbracket\phi\rrbracket_H(x) = 0$$

On this ground, the uniform preservation theorem given in the previous section (cfr. Theorem 1) applies to our specialized semantics, as stated in Corollary 2.

Corollary 2. *Let A be a modal abstraction for the hybrid automaton H, and assume to interpret μ-calculus formulas according to Definition 11. Then $A \models_3 \phi \preceq H \models \phi$.*

We conclude this subsection by providing a concrete example, which illustrates our three-valued semantics on modal abstractions.

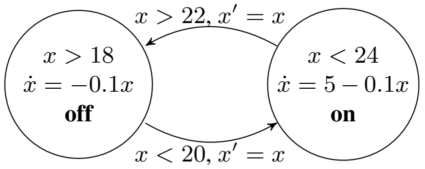


Fig. 3. Heating Controller

Figure 3 depicts a heating controller consisting of the two discrete states **off** and **on**. While the heating is off, the temperature x falls via the differential rule $\dot{x} = -0.1x$. Conversely, while the heating is on, the temperature rises via $\dot{x} = 5 - 0.1x$. The location **off** may be left, when the temperature falls below 20 degree and it has to be left, when x falls below 18 degree. Symmetric conditions hold for **on**. Initially, the heating controller starts at the location **off** with a temperature of 20 degrees. Figure 1 and Figure 2 depict two different modal abstractions A_1 and A_2 for the heating controller. Consider the formula $\psi = \mu Z.\phi \vee \Diamond Z$, where ϕ denotes a propositional letter being true for the abstract state $(\textbf{off}, (20, 24))$. This formula holds in the states that can reach a configuration where the temperature is between 20 and 24 degree and the heating is off. Applying the semantics scheme on A_1, this formula can not be proven since A_1 does not admit a must-path from the initial region to $(\textbf{off}, (20, 24))$. Conversely, ψ can not be falsified, since there exists a may-path from the initial region to the target region. Using A_2 instead we can establish $A_2 \vDash \psi$, since A_2 contains a must-path leading to $(\textbf{off}, (20, 24))$. This yields $H \vDash \psi$, by our preservation theorem.

5.2 Semantics Completion for DBB Abstractions

We now turn out to the consideration of DBB abstractions, providing a further specialization of the uniform preservation result discussed in section 4.

DBB abstractions encode the information for parallel over- and underapproximated reachability analysis differently from modal abstractions. In particular, there is no distinction between edges that over-estimate (resp. under-estimate) the evolution of the underlying hybrid automaton. Rather, a discrete edge between the abstract states $[r]_{\equiv_n}$ and $[r]_{\equiv_n}$ in $H_{\equiv_n}$ means that H can evolve from $[r]_{\equiv_n}$ to $[r']_{\equiv_{n-1}} \supseteq [r']_{\equiv_n}$, via a discrete edge. The continuous edges in $H_{\equiv_n}$ represent instead with fidelity the continuous evolution along the regions of the abstraction. These considerations are useful to understand the ratio underlying the development of the exact semantics for the modal operators on DBB abstractions, given in Definition 12.

Definition 12. *Let H be a hybrid automaton and $H_{\equiv_n} = \langle Q_{/\equiv n}, Q_{0/\equiv n}, l_{\rightarrow}, \rightarrow_{/\equiv n}\rangle$ be its n-DBB abstraction. Then the semantics scheme in Definition 10 is completed by the following rules:*

- *The value of $[\![\langle\delta\rangle\varphi]\!]_{\equiv n}([x]_{\equiv n})$ is given by*

$$\begin{cases} 1 & \exists [x']_{\equiv n} \in Q_{/\equiv n} : [x]_{\equiv n} \xrightarrow{\delta} [x']_{\equiv n} \wedge [\![\varphi]\!]_{\equiv n}([x']_{\equiv n}) = 1 \\ 0 & \nexists [x']_{\equiv n} \in Q_{/\equiv n} : [x]_{\equiv n} \xrightarrow{\delta}{}^{*} [x']_{\equiv n} \wedge [\![\varphi]\!]_{\equiv n}([x']_{\equiv n}) = 1 \\ \bot & otherwise \end{cases}$$

- *The value of $[\![\langle e\rangle\varphi]\!]_{\equiv n}([x]_{\equiv n})$ is given by*

$$\begin{cases} 1 & \exists [x']_{\equiv n} \in Q_{/\equiv n} : [x]_{\equiv n} \xrightarrow{\delta} [x']_{\equiv n} \wedge [\![\varphi]\!]_{\equiv n-1}([x']_{\equiv n}) = 1 \\ 0 & \nexists [x']_{\equiv n} \in Q_{/\equiv n} : [x]_{\equiv n} \xrightarrow{e} [x']_{\equiv n} \wedge [\![\varphi]\!]_{\equiv n}([x']_{\equiv n}) \neq 0 \\ \bot & otherwise \end{cases}$$

- $[\![[a]\varphi]\!]_{\equiv n} := [\![\neg(\langle a\rangle\neg\varphi)]\!]_{\equiv n}$ *for* $a \in \{e, \delta\}$
- *Let us use the notation $\{[x_i]_{\equiv n}\}$ to represent an infinite path in $H_{\equiv n}$. Then: The value of $[\![E(\varphi \underline{U} \psi)]\!]_{\equiv n}([x_0]_{\equiv n})$ is given by*

$$\begin{cases} 1 & \exists\{[x_i]_{\equiv n}\}\exists k \in \mathbb{N} : 1.\ [x_{i<k}]_{\equiv n} \xrightarrow{\delta} [x_{i+1}]_{\equiv n} \wedge [\![\varphi \vee \psi]\!]_{\equiv n}([x_i]_{\equiv n}) = 1 \\ & \qquad\qquad\qquad 2.\ [\![\psi]\!]_{\equiv n}([x_k]_{\equiv n}) = 1 \textit{ or} \\ & \qquad\qquad\qquad\quad [x_k]_{\equiv n} \xrightarrow{e} [x_{k+1}]_{\equiv n} \wedge [\![E(\varphi \underline{U} \psi)]\!]_{\equiv n-1}([x_{k+1}]_{\equiv n-1}) = 1 \\ 0 & \forall\{[x_i]_{\equiv n}\}\forall k \in \mathbb{N} : [\![\psi]\!]_{\equiv n}([x_k]_{\equiv n}) \neq 0 \Rightarrow \exists j < k :\ [\![\varphi \vee \psi]\!]_{\equiv n}([x_j]_{\equiv n}) = 0 \\ \bot & otherwise \end{cases}$$

The value of $[\![A(\varphi \underline{U} \psi)]\!]_{\equiv n}([x_0]_{\equiv n})$ *is given by*

$$\begin{cases} 1 & \forall\{[x_i]_{\equiv n}\}\exists k \in \mathbb{N} : [\![\psi]\!]_{\equiv n}([x_k]_{\equiv n}) = 1 \ \wedge \ [\![\varphi \vee \psi]\!]_{\equiv n}([x_{i<k}]_{\equiv n}) = 1 \\ 0 & \exists\{[x_i]_{\equiv n}\}\exists k \in \mathbb{N} : 1.\ [x_{i<k}]_{\equiv n} \xrightarrow{\delta} [x_{i+1}]_{\equiv n} \wedge [\![\varphi \wedge \neg\psi]\!]_{\equiv n}([x_i]_{\equiv n}) = 1 \\ & \qquad\qquad\qquad 2.\ [\![\varphi \vee \psi]\!]_{\equiv n}([x_k]_{\equiv n}) = 0 \textit{ or} \\ & \qquad\qquad\qquad\quad [x_k]_{\equiv n} \xrightarrow{e} [x_{k+1}]_{\equiv n} \wedge [\![A(\varphi \underline{U} \psi)]\!]_{\equiv n-1}([x_{k+1}]_{\equiv n-1}) = 0 \\ \perp & \textit{otherwise} \end{cases}$$

On the ground of Lemma 3 the uniform preservation theorem in Section 4 applies also to our specialized semantics for DBB abstractions, as formally stated in Corollary 3.

Lemma 3. *Let $H_{\equiv n}$ be an n-DBB abstraction for the hybrid automaton H, and assume to interpret μ-calculus formulas according to Definition 12. Then, for any formula $\phi \in \{\langle\delta\rangle\varphi, \langle e\rangle\varphi, [\delta]\varphi, [e]\varphi, E(\varphi \underline{U} \psi), A(\varphi \underline{U} \psi)\}$:*

$$[\![\phi]\!](r) = 1 \Rightarrow \ \forall\, x \in r : \ [\![\phi]\!]_H(x) = 1$$
$$[\![\phi]\!](r) = 0 \Rightarrow \ \forall\, x \in r : \ [\![\phi]\!]_H(x) = 0$$

Corollary 3. *Let $H_{\equiv n}$ be an n-DBB abstraction for the hybrid automaton H, and assume to interpret μ-calculus formulas according to Definition 12. Then for any formula $\phi \in L_\mu$: $H_{\equiv_n} \models_3 \phi \preceq H \models \phi$*

The following example illustrates the instantiation of the semantic framework to DBB abstractions described so far.

The hybrid automaton depicted in Fig. 4 models a water level controller with two variables. The first variable x_1 represents a clock, while the second variable x_2 models the water level in the tank. When the valve at the bottom of the tank is closed, the water level increases by $1ms^{-1}$, otherwise it decreases by $2ms^{-1}$. Intuitively, the clock allows to establish that the valve remains open as long as it was closed in the previous step. This hybrid automaton does not belong to any known decidable class , and it yields infinite bisimulations for suitable initial partitions [15]. This is the case e.g. for the initial partition

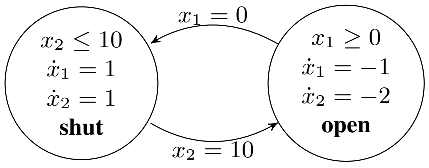


Fig. 4. Water Level Controller

$$P = \{\textbf{shut} \times X,\ \textbf{shut} \times Y,\ \textbf{open} \times X,\ \textbf{open} \times Y\}$$

where $X = [0, 6] \times \{10\}$ and $Y = [0, \infty) \times (-\infty, 10] \setminus X$. However, the above automaton is fully O-minimal and thus the construction of DBB-Abstractions terminates [10].

Consider the following question 'When starting in *Init* = **open** $\times [0, 6] \times \{10\}$, does the water level controller always admit an evolution to r = **shut** $\times [0, 6] \times \{10\}$?'. Such a question corresponds to compute whether $H \models \psi$, where $\psi = \mu Z.r \vee \Diamond Z$. We use DBB abstractions to falsify the above property. Figure 5 and Fig. 6 depict the 0-DBB and 1-DBB abstraction, respectively. In the 0-DBB abstraction the formula ψ evaluates to 1 on r_1, r_2, r_3 and s_1, and is indefinite elsewhere. Thus, $(H \models \psi) = \perp$ since $[\![\psi]\!](s_2) = \perp$ for the only initial region s_2 of $H_{\equiv 0}$. In the 1-DBB abstraction $H_{\equiv 1}$ the region s_2 gets split to $\langle t_2, t_3\rangle$ and ψ evaluates to 0 on t_3. Since all a paths starting in t_3 do not allow to reach a region, where ψ evaluates to 1 or $\perp$, we can conclude that $(H_{\equiv 1} \models \psi) = 0$. Thus, due to the preservation theorem we can state that $H \not\models \psi$.

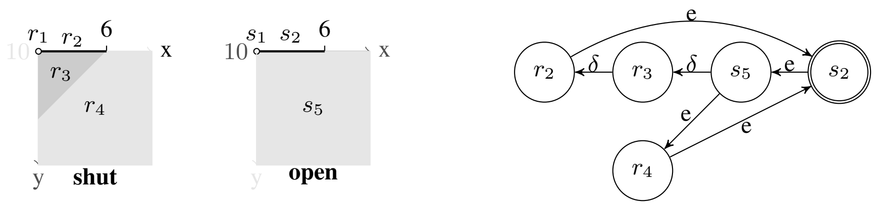


Fig. 5. 0-DBB Abstraction: Partitioning of the Regions and Control Graph of the Abstraction (for simplification the cycle $r_1 \leftrightarrow s_1$ is left out)

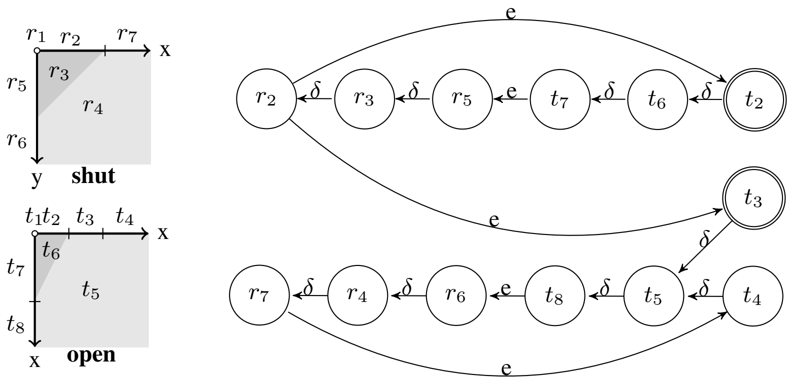


Fig. 6. 1-DBB Abstraction: Partitioning of the Regions and Control Graph of the Abstraction (for simplification the cycle $r_1 \leftrightarrow s_1$ is left out)

6 Abstraction Refinement and Monotonicity

A key issue in the context of three-valued abstract semantics for μ-calculus on hybrid automata is related to *monotonicity*. Given an abstraction-refinement framework, it is desirable that the set of formulas evaluating to $\perp$ decreases monotonically in its size along any succession of finer abstractions. Such a requirement is reminiscent of the usual *regularity* property for Kleene's three-valued logics [8,19].

In this section, we compare the two abstraction refinement frameworks based on DBB-abstractions and modal abstractions with respect to monotonicity. Theorem 2 proves that the DBB succession of abstractions allows to monotonically recover true/false μ-calculus formulas along the series of refining abstractions.

Theorem 2 (Monotonicity). *Let $H_{\equiv n}$ and $H_{\equiv k}$ with $n > k$ be DBB abstractions of the hybrid system H, and let ϕ be a μ-calculus formula. Then, $(H_{\equiv k} \models \phi) \preceq (H_{\equiv n} \models \phi)$.*

The following example shows instead that the abstraction/refinement framework based on modal abstractions does not behave well with respect to monotonicity.

Example 1. Let us consider the abstraction A_3 depicted in Fig. 7 which is a refinement of the abstraction A_2 given in Fig. 2. In Section 5.1 we were able to establish the result

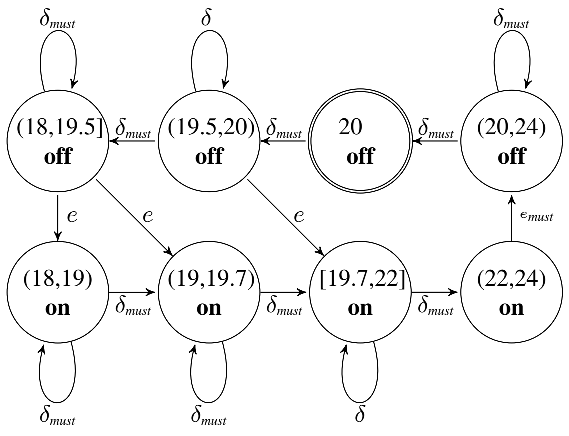


Fig. 7. Abstraction A_3 with may/must of the heating controller

$(A_2 \models_3 \mu Z.\phi \vee \Diamond Z) = 1$, where ϕ is a propositional letter being true on $(20, 24) \times$ **off**. However, we cannot prove $(A_3 \models_3 \mu Z.\phi \vee \Diamond Z) = 1$ since there exist no $\xrightarrow{e}_{must}$-transitions from the configuration **off** to the configuration **on**.

7 Conclusions

In this paper, we developed a framework for inferring general μ-calculus properties on abstractions of hybrid automata. Based on the definition of a sound three-valued semantics on abstractions, our framework does not feature the inherent limitations of bounded model checking or techniques using the simulation preorder. In particular, our method can *both* prove and disprove arbitrary μ-calculus properties on abstractions over- and underapproximating (unbounded) evolutions of the system. To cope with the variety of candidate abstractions for our framework, we rely on a top-down approach in which we (1) fix the semantics of boolean and fixpoint operators while only constraining the modal operators, and (2) consider suitable classes of abstractions to instantiate the modal operators according to the constraints. We finally show that, despite of the generality of the preservation result, the choice of the abstraction is relevant for the *monotonic* preservation of true/false evaluations along abstraction refinements.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., Henzinger, T.A., Ho, P.: Automatic symbolic verification of embedded systems. In: IEEE Real-Time Systems Symposium, pp. 2–11 (1993)
3. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. of the IEEE 88, 971–984 (2000)
4. Bauer, K.: Three-valued μ-calculus on hybrid automata. Master's thesis, Master Thesis, University of Kaiserslautern, Department of Computer Science (2008)

5. Bensalem, S., Bouajjani, A., Loiseaux, C., Sifakis, J.: Property preserving simulations. In: von Bochmann, G., Probst, D. (eds.) CAV 1992. LNCS, vol. 663, pp. 260–273. Springer, Heidelberg (1993)
6. Davoren, J.: On hybrid systems and the modal μ-calculus. In: Antsaklis, P.J., Kohn, W., Lemmon, M.D., Nerode, A., Sastry, S.S. (eds.) HS 1997. LNCS, vol. 1567, pp. 38–69. Springer, Heidelberg (1999)
7. Davoren, J., Nerode, A.: Logics for hybrid systems. Proc. of the IEEE 88, 985–1010 (2000)
8. Fitting, M.: Kleene's three valued logics and their children. Fund. Inf. 20, 113–131 (1994)
9. Fränzle, M.: What will be eventually true of polynomial hybrid automata? In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 340–359. Springer, Heidelberg (2001)
10. Gentilini, R., Schneider, K., Mishra, B.: Successive abstractions of hybrid automata for monotonic CTL model checking. In: Artemov, S.N., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 224–240. Springer, Heidelberg (2007)
11. Ghosh, R., Tiwari, A., Tomlin, C.: Automated symbolic reachability analysis with application to delta-notch signaling automata. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 233–248. Springer, Heidelberg (2003)
12. Ghosh, R., Tomlin, C.J.: Lateral inhibition through delta-notch signaling: A piecewise affine hybrid model. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 232–245. Springer, Heidelberg (2001)
13. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
14. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. of 36th Ann. Symp. on Found. of Comp. Sc., p. 453. IEEE, Los Alamitos (1995)
15. Henzinger, T.: Hybrid automata with finite bisimulations. In: Fülöp, Z., Gecseg, F. (eds.) ICALP 1995. LNCS, vol. 944, pp. 324–335. Springer, Heidelberg (1995)
16. Henzinger, T.A.: The theory of hybrid automata. In: Proc. of the 11th IEEE Symp. on Logic in Comp. Science, pp. 278–292. IEEE Computer Society, Los Alamitos (1996)
17. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proc. of the 27th Symp. on Theory of Computing, pp. 373–382. ACM, New York (1995)
18. Kannellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation 86(1), 43–68 (1990)
19. Kleene, S.C.: Introduction to Metamathematics. Wolters-Noordhoff, Groningen (1971)
20. Lafferriere, G., Pappas, G., Sastry, S.: O-minimal hybrid systems. Mathematics of Control, Signals, and Systems 13, 1–21 (2000)
21. Lafferriere, G., Pappas, J., Yovine, S.: A new class of decidable hybrid systems. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 137–151. Springer, Heidelberg (1999)
22. Miller, J.: Decidability and complexity results for timed automata and semi-linear hybrid automata. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 296–309. Springer, Heidelberg (2000)
23. Piazza, C., Antoniotti, M., Mysore, V., Policriti, A., Winkler, F., Mishra, B.: Algorithmic algebraic model checking i: Challenges from systems biology. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 5–19. Springer, Heidelberg (2005)
24. Alur, C.C.R., Henzinger, T.A.: Computing accumulated delays in real-time systems. Formal Methods in System Design 11, 137–156 (1997)
25. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 573–589. Springer, Heidelberg (2005)
26. Tiwari, A., Khanna, G.: Series of abstractions for hybrid automata. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 465–478. Springer, Heidelberg (2002)

Automatic Boosting of Cross-Product Coverage Using Bayesian Networks

Dorit Baras, Laurent Fournier, and Avi Ziv

IBM Research Laboratory in Haifa, Israel
{doritb,laurent,aziv}@il.ibm.com

Abstract. Closing the feedback loop from coverage data to the stimuli generator is one of the main challenges in the verification process. Typically, verification engineers with deep domain knowledge manually prepare a set of stimuli generation directives for that purpose. Bayesian networks based CDG (coverage directed generation) systems have been successfully used to assist the process by automatically closing this feedback loop. However, constructing these CDG systems requires manual effort and a certain amount of domain knowledge from a machine learning specialist. We propose a new method that boosts coverage at early stages of the verification process with minimal effort, namely a fully automatic construction of a CDG system that requires no domain knowledge. Experimental results on a real-life cross-product coverage model demonstrate the efficiency of the proposed method.

1 Introduction

Functional verification remains one of the main challenges in the hardware design cycle [1]. In current industry practice, dynamic verification is the leading technique for functional verification. To cope with the ever increasing complexity of modern designs, the verification process is a highly automated process relying on sophisticated tools to replace the human engineer in almost every aspect of operating the verification environment, such as generating stimuli for the *design under verification* (DUV), and checking that the DUV behavior is according to its specification [1].

Functional coverage is the main technique for checking that the verification has been thorough [2]. Coverage can help monitor the quality of verification and direct the verification team toward areas that have not been adequately verified. The analysis of coverage information and the use of this information to direct the stimuli generator toward uncovered or lightly covered areas is one of the remaining human bottlenecks in today's verification environment. Therefore, considerable effort is spent finding ways to automate the covering procedure; that is, to close the loop of coverage analysis and stimuli generation. Although in early stages of the verification process, reaching 100% coverage is not the main priority of the verification team, reaching a high level of coverage as fast as possible is important because bugs found in the early stages of the design require far less time and effort to fix. Consequently, it would be helpful to boost coverage in the early stages of verification with minimal effort from the verification team. This requirement motivated the work presented in this paper.

Data-driven Coverage directed test generation (CDG) is a technique to automate the feedback from coverage analysis to stimuli generation. In data-driven CDG, the CDG system discovers the relations between the directives that control the stimuli generation and the coverage events, based on observations of specific settings of the directives and the coverage events to which they lead. Reports on several CDG systems of this kind have been published in recent years, including systems based on Bayesian networks [3,4], Markov chains [5], genetic algorithms [6], and inductive logic [7].

Bayesian networks [8] are well suited to address the challenges of data-driven CDG and provide the kind of modeling required for CDG. First, Bayesian networks offer a natural and compact representation of the rather complex relationship between the CDG ingredients, namely, the coverage events on the one hand and the test directives on the other. In addition, Bayesian networks provide the ability to encode essential domain knowledge in the CDG system. As a result, Bayesian network CDG systems were able to produce high coverage and high coverage rate in several industrial settings [3,4,9,10].

The CDG approaches mentioned above require a certain amount of domain knowledge of an expert familiar with the design details and an expert in Machine Learning. In addition, construction of the CDG system may require significant effort that cannot be invested in early stages of the verification process. We propose a fully automated method for constructing CDG systems based on Bayesian networks, which does not require domain expertise. In contrast to existing works using Bayesian networks [3,4], we suggest a process that does not require stages of pre-processing or help from either verification engineers or machine learning specialists. Similarly to the manual construction process described in [3], the components of the automated process include a stage of selecting relevant directives, followed by construction of the Bayesian network and learning its parameters. Once this is done, the network can be used to tune the directives in order to reach desired coverage events. Defining the structure of the Bayesian network is the most difficult part in the manual process. In the automated process, we replace the manual construction with several generic structure learning algorithms [11,12,13,14]. We then enhance these algorithms with two heuristics that suit the characteristics of the CDG setting, namely, pruning edges between directives and quantizing the probabilities of directives. These automatically constructed networks may not be as good as the manually constructed ones, but they do provide enough power to achieve the goals of coverage boosting.

We tested our method on a cross-product coverage model used in the verification of the instruction fetch unit (IFU) of the IBM z10 processor. Our results indicate that the automatic booster approach is working. We were able to achieve significant improvement in coverage over the regression suite used in the verification process with a fully automated process that requires minimal effort. We also present results showing that the structure learning algorithms perform well with a noisy set of parameters. Moreover, the two heuristics we propose improve

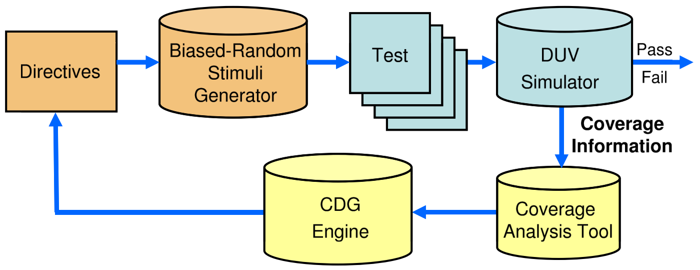


Fig. 1. Structure of a verification environment with CDG

the quality of the learned Bayesian networks and perform better than the generic structure learning algorithms.

The rest of the paper is organized as follows: Section 2 provides details on Bayesian networks and their application to coverage directed generation. In Section 3, we explain the concept of the coverage booster. Section 4 describes the fully automatic process of the coverage booster. Section 5 describes our experimental results. We conclude and present directions for future work in Section 6.

2 Coverage Directed Generation Using Bayesian Networks

In the highly automated verification environment used today, analysis of coverage information and usage of this information to direct the stimuli generator toward uncovered or lightly covered areas is one of the remaining human bottlenecks. Therefore, considerable effort is spent on finding ways to automate the covering procedure–that is, to close the loop of coverage analysis and stimuli generation. This automated feedback from coverage analysis to stimuli generation, known as *Coverage Directed stimuli Generation* (CDG), can reduce the manual work in the verification process and increase its efficiency. In general, the goal of CDG is to automatically provide the stimuli generator with directives that are based on coverage analysis [3]. Figure 1 presents a sketch of a verification environment with CDG. The CDG engine receives information from the coverage analysis tool about the state and progress of the coverage, and generates directives to the random test generator that are designed to achieve one or many of the CDG goals.

There are two main approaches to CDG. In direct CDG, or model-based CDG, an external model of the design under verification is used to generate stimuli to accurately hit the coverage events [15]. In data-driven CDG, which is often called feedback-based CDG, the CDG system relies on inference of the required stimuli directives from observations of past behaviors [3]. This inference is usually done with machine learning techniques [3,6,7]. In this paper, we refer to CDG systems based on Bayesian networks for cross-product coverage [3]. The rest of this section explains how such systems are constructed and used.

2.1 A Brief Introduction to Bayesian Networks

A Bayesian network is a graphical representation of the joint probability distribution for a set of variables. It consists of two components. The first is a directed acyclic graph in which each vertex corresponds to a random variable. This graph represents a set of conditional independence properties of the represented distribution: each variable is independent of its non-descendants in the graph, given the state of its parents. The graph captures the qualitative structure of the probability distribution and is exploited for efficient inference and decision making. The second component is a collection of local interaction models that describe the conditional probability $p(X_i|Pa_i)$ of each variable X_i given its parents Pa_i. Together, these two components represent a unique joint probability distribution over the complete set of variables X [8]. The joint probability distribution is given by $p(X) = \prod_{i=1}^{n} p(X_i|Pa_i)$. It can be shown that this equation actually implies the conditional independence semantics of the graphical structure given earlier. The equation above shows that the joint distribution specified by a Bayesian network has a factored representation as the product of individual local interaction models. Thus, while Bayesian networks can represent arbitrary probability distributions, they provide a computational advantage for those distributions that can be represented with a simple structure.

Typical types of queries that can be efficiently answered by the Bayesian network model are derived from applying the Bayes rule to yield posterior probabilities for the values of a node (or set of nodes), X, given some evidence, E, i.e. assignment of specific values to other nodes: $p(X|E) = \frac{p(E|X)*p(X)}{p(E)}$. Thus, a statistical inference can be made in the form of either selecting the *Maximal A Posteriori* (MAP) probability, $\max p(X|E)$, or obtaining the *Most Probable Explanation* (MPE), $\arg\max p(X|E)$.

2.2 A Bayesian Network for CDG

The idea behind using Bayesian networks for CDG cross-product coverage models starts from the understanding that the space containing the directives to the stimuli generator on one side and the coverage model on the other side is a large distribution space. On one side of this space stand the directives to the stimuli generator, which are defined as probability distributions over a domain of values. On the other side stands the cross-product coverage model, which is represented by its attributes. Activating the verification environment[1] with different settings of the directives, or even with the same settings but different random seed, yields different coverage events. Therefore, the coverage attributes can also be viewed as random variables.

Bayesian networks can represent this large distribution space in a compact form. The structure of the network captures the true dependencies between the various components of the space. Specifically, it captures directives that directly affect the values of specific coverage attributes and dependencies between the

[1] That is, generating stimuli using the directives settings, simulating the DUV, and obtaining coverage data.

values of various coverage attributes. Once the structure and parameters of the Bayesian network are defined, an *abductive* query that provides evidence on the effect nodes (desired coverage events) can be used to determine the possible cause (directives settings).

Constructing a Bayesian network for a CDG system comprises three main steps. The first step is selecting the relevant directives to be used in the system. The number of directives is narrowed down to avoid networks that are too large for training and inference. It also reduces the amount of data required for the learning process that follows. The second step, which is the most difficult one, is defining the network structure. In most applications of Bayesian networks, the structure of the network is defined manually by a domain expert. Although structure learning algorithms exist (e.g., [11,16]) their performance is usually inferior to manually constructed networks. The last step is estimating the conditional probabilities of each node in the network. There are efficient algorithms that can learn these parameters from observations on data in the form of values to (some or all) the network nodes. In the CDG case, the set of observations is a sample set of directive settings along with their coverage events as resulting from activating the simulation environment. The constructed Bayesian network can be used in a CDG system to determine directives for a desired coverage event.

It is important to note that the usage of Bayesian networks for CDG is different from most "classical" uses. These characteristics are caused by the way stimuli generators and verification environments behave. In many cases, stimuli generators use the settings of the same directive many times during their operation, each time randomly choosing a different value according to the distribution specified in the settings. This allows them to generate rich sequences of values out of one directive. Therefore, it is important to provide the Bayesian network and the algorithms that learn it with settings that specify probability distributions on the possible values of a directive, not just settings that determinately define its value. We call such settings *soft evidence*. The implications of using soft evidence in the automatic construction of a Bayesian network are discussed in Section 4.

3 The Coverage Booster

Coverage closure is commonly acknowledged as one the most important goals of the verification process. The recurring observation in this domain is that a portion of the events to be covered are not reached through the initial attempts. These events are also usually some of the most complex and subtle events, as they resisted the preliminary regular attempts to create them. Therefore, this involved task is usually carried out by experts, both in the application domain and in the stimuli generation technology.

Due to the costly price of the coverage closure task, both in terms of time investment and expertise required, much attention has been steered toward the attempt to insert some automation into this classically manual process. Our approach, relying on Machine Learning techniques [3], has shown good initial signs indicating the potential for automating this task. Indeed, there were several success stories [4,10] that demonstrated the adequacy of this approach, and showed that this apparently intrinsically manual task could be modeled and performed by a program.

While machine learning techniques have shown their capacity to capture part of the compound relationship between events and stimuli generation directives and automate the process of closing the loop between stimuli generation directives and coverage events, constructing the CDG system is still a manual process requiring both expertise and effort. This significantly limits the opportunities of CDG to provide real benefits to the verification process. First, CDG is beneficial only in places where it replaces significant effort in coverage closure. This usually means that there are either large and complex coverage models [10], or extremely important coverage events [4]. In addition, CDG is possible only when a significant effort is directed at coverage closure. In many cases, this happens only towards the end of the verification effort.

"Coverage Booster" is our new approach for exploiting the demonstrated capacity of machine learning in this domain, while showing an improvement, or boost, in efficiency measured in overall human effort. Instead of placing full coverage as a primary goal, we target minimal human effort above any other consideration. The coverage booster may not reach full coverage, but because of the zero human effort spent, covering new events and improving the coverage rate are enough to create real benefits to the verification process. Therefore, the coverage booster expands the envelope of opportunities for CDG in two ways. First, CDG can be useful for more coverage models for which the verification team cannot allocate much effort. In addition, CDG can be used in earlier stages of the verification process, before coverage becomes top priority.

4 Automatic Construction of CDG Engine

At the heart of a Bayesian network-based CDG engine lies a Bayesian network that allows a compact yet accurate description of the stochastic space of the coverage attributes on the one side and the stimuli generator directives on the other side. As noted previously, the goals of our coverage booster are to eliminate the need for manual intervention and boost coverage in early stages of the verification process. To meet these goals, the Bayesian network needs to be created automatically. The automated process described in this section generally follows the manual process described in [3], except that the manual steps requiring domain expertise in either the DUV and verification environment or machine learning techniques are replaced with automated steps. This enhanced automation may lower the quality of the Bayesian network and prevent it from reaching coverage closure. However, it does provide enough power to achieve the goal of coverage boosting.

The automated process is composed of three stages: feature selection, structure learning, and parameter learning. Figure 2 illustrates the automatic process that creates a Bayesian network, which is later used for coverage boosting. Initially, the verification team provides a description of the coverage model and

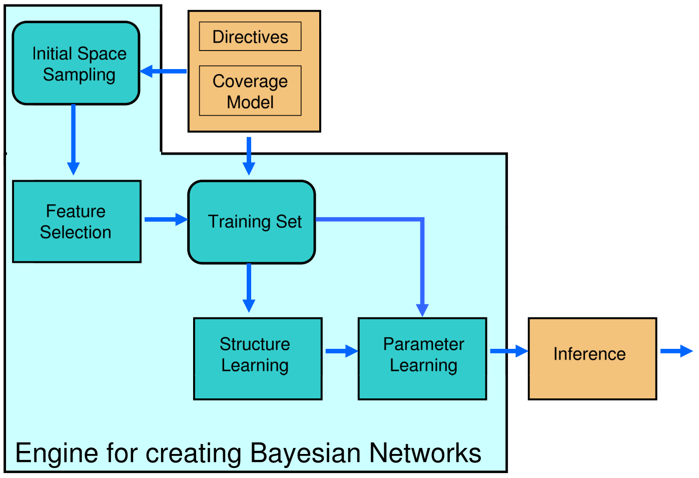


Fig. 2. Construction of the Bayesian networks for CDG

a list of directives that control the stimuli generator. The number of directives needs to be cut to allow efficient learning. In order to perform this selection we create an initial sampling by simulating with random directives. This is followed by a feature selection phase in which we narrow down the number of directives that are used later; we do this by choosing those with maximal influence. Next, we create a training set with random sampling of the directives we chose, while the rest of the directives are given default values. This training set is the input to the structure learning procedure that is followed by a parameter learning phase in which the conditional probabilities of the variables are estimated. At this point we have a Bayesian network that can be used in a CDG engine.

4.1 Feature Selection

A typical verification environment contains tens or even hundreds of directives, which control the stimuli generator and affect the stimuli it generates. Learning with such a large set of directives is very difficult or even impossible. Therefore, it is essential to narrow down the number of directives and select a small high quality subset for the CDG process. It turns out that in most cases only a small subset of the directives are required to control the coverage of a specific model, so this reduction does not reduce the ability of the coverage booster to achieve its goal.

Feature selection is a general term given to a variety of algorithms that extract the subset of features most relevant for a given task [17,18]. Most algorithms perform feature selection on a feature space that is fully observed, i.e., the values of all features are known for every sample. For CDG, the features are directives and the feature selection task involves finding directives that highly influence the coverage attributes. Feature selection in the CDG settings has several unique characteristics that make it difficult. First, the data is very noisy because of the highly stochastic nature of the simulation environment. Second, dominant values that are present in some of the coverage attributes can mask a dependency between a directive and a coverage attribute. The third, and most difficult issue, is the fact that the data is not fully observed. We are not given the exact values of the directives, but rather a distribution over them. Because of these unique characteristics, commonly used algorithms cannot be applied as is.

To overcome these problems, we use *mutual information* [19] as a criterion that could be estimated from a noisy, not fully observed sample set. Mutual information in information theory is a quantity that measures the information between two variables. Let $P(X)$ and $P(Y)$ be the marginal distributions of X and Y respectively and let $P(X, Y)$ be the joint distribution over the two variables. The mutual information between the variables is given by $I(X;Y) = \sum_{x,y} P(x, y) \log \frac{P(x,y)}{P(x)P(y)}$.

We use mutual information as a means of measuring the strength of correlation between coverage attributes and directives. The mutual information is estimated from the statistics of the collected data using a standard maximum-likelihood estimator. We calculate the mutual information of all the directives coverage attributes pairs and select directives if their mutual information with any of the attributes is above a certain threshold, that can be changed in order to reach a desired number of directives. Apart from this feature selection procedure, the pruning heuristics of the structure learning algorithm described in Section 4.2 can also be used for feature selection.

4.2 Structure Learning

In our CDG systems, the relations between the directives and coverage attributes are modeled using a Bayesian network, that is composed of two components: the structure and the parameters. The structure is a DAG that captures causal relations and the parameters represent the conditional probability tables that describe the distribution of a node given its parents. While there are applications where structure learning algorithms provide high quality results (e.g., [10,16]), in the general case, these algorithms have many weaknesses that cause them to be highly inferior to manually constructed structures. Some of these weaknesses are closely related to the unique requirements of Bayesian network for CDG. One example is the use of soft evidence, which is known to produce a richer set of events on the one hand, but makes structure learning more difficult on the other hand. Another example is the inability to construct networks with hidden nodes, which proved to be essential in past CDG work ([3]). Therefore, we do not expect the output of structure learning algorithms to be as good as manually constructed networks. Still, our results show that automatic structure learning can provide structure that is sufficient for coverage boosting.

There are two main approaches to structure learning: generic algorithms that can be applied to any problem satisfying predefined conditions and application-based algorithms that suit the specific setting of given problems (usually with unique domains or characterization). We are not aware of application-specific algorithms that fit the CDG setting; therefore we use several well known generic algorithms: K2 [11], mcmc [12], gs [13] and structural EM [14]. These algorithms are used in a wide range of applications with various levels of success.

The initial experiments with these algorithms provided reasonable results. However, we wanted to improve the results by taking into account the specific nature of CDG settings. For that purpose, we use two techniques. The first is a post-processing heuristic that prunes arcs between directives in the network graph. Our initial learned networks have arcs between directive nodes, for valid reasons. First, combinations of assignments to directives cause simulations to fail or produce small sets of coverage events resulting in dependencies between directives in the training set. In addition, the random independent sampling of the directive space for the training set may still contain undesired dependencies. Still, in the CDG settings, these edges are not desirable because we have total control on the directives settings throughout the process. Moreover, these edges may actually interfere with learning the real important relations between coverage attributes and directives. Therefore, we decided to investigate removing these arcs. The resulting networks, which we refer to as *pruned networks*, yields better results (see Section 5). The same reasoning cannot be used for edges connecting two attribute nodes because they capture dependencies that are essential for high quality networks. Note that removing the edges between directives can result in directives nodes that are disconnected from the attributes nodes as indeed happened in our experiments. These orphan directives are no longer part of the CDG system, therefore, the pruning heuristics can be used as a second order feature selection procedure.

The second improvement to the generic learning algorithm is a pre-processing step that modifies the domain of the directives in order to enrich the settings of directives and overcome the soft vs. hard evidence problem. Hard evidence is the term we use when we assign a single value to a directive. It is like having soft evidence with one probability set to one and all others to zero. In verification, directive nodes are used many times in a single simulation run and in each use a new value is randomly chosen based on the probability specified. Therefore, hard evidence, which forces a single value to each directive, strongly limits the stimuli sequences generated and soft evidence that sets the probabilities for each value in the domain of the directives is essential. However, the structure learning algorithms we use are not capable of handling soft evidence. In order to deal with this, we suggest a heuristic that allows us to incorporate soft evidence in the existing algorithms. We refer to this new method as *quantized directives.*

The main idea of this method is to replace each directive node Y whose domain is $(y_1, \ldots, y_n)$ with node $\tilde{Y}$ whose domain is a discrete set of probabilities over the domain of the original directive Y. For example, if Y has a domain (y_1, y_2, y_3), we can create a node $\tilde{Y}$ with seven values representing the probabilities $\{(1, 0, 0),$ $(0, 1, 0)$, $(0, 0, 1)$, $(\frac{1}{2}, \frac{1}{2}, 0)$, $(\frac{1}{2}, 0, \frac{1}{2})$, $(0, \frac{1}{2}, \frac{1}{2})$, $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})\}$ over (y_1, y_2, y_3). Using this setting allows us to run simulations with soft evidence that enables the full richness of the generated sequences on one hand, and use the corresponding value in $\tilde{Y}$ as hard evidence in the structure learning algorithm. Once the structure is learned, we return to the original directives for the parameter learning stage. This quantization procedure is not limited to uniform values over subsets. It can include any soft evidence chosen. However, there is a trade-off between the ability to use many soft evidence values and the domain size of the extended directive. Domain sizes that are too large will cause the learning to be more difficult and lower the ability of the network to perform inference.

4.3 Parameter Learning

Once the structure of the Bayesian network is known, the next step is to learn the parameters of its nodes, or, in other words, the conditional probability of each node given the values of its parents. Unlike structure learning, there exist algorithms for parameter learning that deals with soft evidence [20]. The algorithm we use is the EM (Expectation-Maximization) algorithm [20]. This is an iterative algorithm that is guaranteed to converge to a local maximum of the likelihood. The EM algorithm learns not only the conditional probabilities of internal nodes, but also the prior probabilities of root nodes (nodes with no parents). When such a node corresponds to a directive, calculating the prior probabilities is not needed or even not desirable for the same reasons that led to the pruning heuristic. In an on going research, we try to generalize the learning algorithms to address this issue.

5 Experiments

We conducted several experiments using the verification environment for the z10 processor's instruction fetch unit (IFU), which is built into the latest IBM System z (mainframe) computers. The IFU is responsible for efficiently requesting and buffering instruction data fetches from memory, and pre-decoding them. One of the most important functions for good unit performance is branch prediction. That is a complex task which includes the prediction of several aspects of a branch. The coverage model used in the experiments looks at a snapshot of the processor pipeline and contains 13 attributes that describe the state of each pipe stage, as well as other flags relating to branches and pipe recycles. The first experiments were conducted during the development stage of the project, and contributed to the quality of the final product.

While the experiments included all steps of the automatic construction of the CDG system described in Section 4, we focused on the structure learning aspect of the construction, which is the most challenging step. The initial set of directives provided by the verification team included 22 directives. Using the feature selection procedure described in Section 4.1, we reduced this set to five directives. For the parameter learning we used the EM algorithm [20].

We tested the ability of four structure learning algorithms to boost coverage over normal activations of the verification environment using its regression suite. The regression suite is a collection of directives settings that are manually designed by the verification team to cover areas of interest. The number of directives used in the regression suite is much larger than the five directives we selected for our experiments or even the initial 22 directives given to us. We started our experiments after about 20,000 simulation runs that covered 5222 events in the model. At that time, we observed a considerable slow down in the coverage rate (see Figure 3). This was done for two main reasons. First, we wanted to avoid dealing with the very easy-to-cover events that are hit anyway and thus are not important to the coverage booster. Second, this setting represents a more realistic setting, where the coverage booster is not operated from day one.

We used an additional two references in the experiments. The first was a large set of randomly created directive settings for the five selected directives. This reference was used for two reasons: First, to verify that the regression suite is not naive and more importantly, to ensure that the feature selection phase alone is not enough for boosting. The last reference was a naive Bayesian network that contained edges between directives and attributes and between pairs of attributes with high mutual information. All the experiments followed the same procedure. We applied a structure learning algorithm and the parameter learning algorithm to obtain a trained Bayesian network. Then we selected 5000 uncovered events and generated (for each network) two sets of directives for each event (using MAP and MPE queries). Finally, we simulated the IFU with the generated directives and measured the corresponding coverage. Note that the networks were not able to produce directives in all the cases, so the total number of simulation runs for each network was smaller than the maximum possible 10,000.

The experiments tested the quality of the four structure learning algorithms mentioned in Section 4.2, namely K2 [11], mcmc [12], gs [13] and structural EM [14]. The experiments tested and compared the basic algorithm and the two heuristics proposed in Section 4.2. In addition, we tested the ability of the structure learning algorithm to perform feature selection. Specifically, the four experiments we conducted were:

Basic algorithm - The goal of this experiment was to test the ability of the structure learning algorithms in their basic form to produce networks that boost coverage. The results of this experiment were also used as a reference to the heuristics in the following experiments.

Pruning - This experiment was aimed at testing the performance of pruned networks heuristic we suggested. We used the networks of the previous experiment, but removed edges between directives. Removing these edges resulted with directive nodes that are orphans, i.e., not connected to any attribute.

Quantization - The goal of this experiment was to test the ability of the quantization heuristic to overcome the soft evidence problem of the structure learning algorithms. Here, we used a subset of three out of the five original directives. These were chosen because they remained connected to attributes in all the pruned networks of the second experiment (except EM that already proved to result in sparse networks).

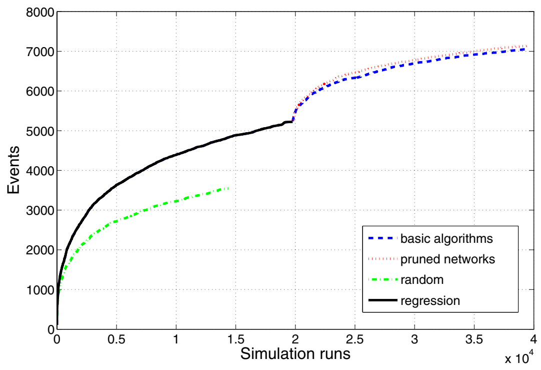


Fig. 3. Coverage progress for random, regression, and aggregated results of basic and prune experiments

Large set of directives - In this experiment, we added to the structure learning algorithm six more directives, with various levels of correlation. The experiment tested the ability of the structure learning with pruning to perform feature selection by leaving orphans nodes.

Figures 3 - 5 and Table 1 summarize the results of the four experiments. Figure 3 shows the coverage progress for the random sampling and the regression suite and the progress of the aggregated results of the basic and pruning experiments. As expected, the random sampling shows the worst behavior over all our settings. More importantly, the figure shows that both the basic algorithms and their pruning enhancement are able to boost the coverage. The next figures and table provide more details on the performance of the four basic algorithms and their enhancements. Figure 4 compares the performance of the four basic algorithms. The figure presents the coverage progress of each of these algorithms and the naive network. The figure shows that all four algorithms provide a better coverage rate than the naive network. This confirms the benefits of the structure learning algorithms, even in their basic form. There is some variation in the performance of the four algorithms. This variation is in the coverage rate they provide, the number of simulation runs they produced out of the 10,000 possible ones, and the number of new events they cover. These differences are also presented in Table 1. The table shows, for each algorithm and each experiment, the number of simulation runs produced and the number of new events covered.

Out the four algorithms, the gs algorithm provides the best results. It produces a high number of simulation runs and it has a high coverage rate. Therefore, although the EM algorithm produces more runs, and K2 and mcmc have a slightly better rate, gs covers the largest number of new events.

The pruning heuristic used in the second experiment not only pruned the edges between directives, it also left some directives orphaned. In fact, one of the directives, which was the weakest among the five selected directives, was

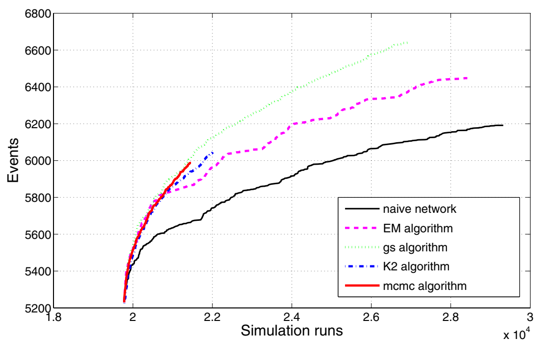


Fig. 4. Coverage progress for the naive network and the four networks in the basic experiment

Table 1. Summary of experimental results

	Basic		Pruning		Quantization		Large Set	
Network	Events	Runs	Events	Runs	Events	Runs	Events	Runs
Naive	970	9545						
K2	824	2233	918	2335	1374	5546	833	2309
gs	1419	7150	1454	6868	1553	9306	1498	10723
mcmc	766	1685	871	1653	919	2356	927	3492
EM	1226	8641	1261	8916				
Aggregate	1837	19709	1918	19772	1936	17208	1713	16524

removed in three out of the four networks. In general, as Table 1 shows, the pruned networks produce roughly the same number of runs as the networks before the pruning and cover 3%-15% more events. The overall improvement of the pruning heuristic over the basic algorithms is also shown in Figure 3 and Figure 5. The quantization technique allows the structure learning algorithm to use soft evidence. The results of this experiment show that even with a limited amount of softness that was used in the experiment, the networks were able to produce many more runs (twice as many for K2) and reach many more events (more than 50% for K2). The EM algorithm failed to learn a network in this experiment and therefore did not produce any results.

The attempt to use the structure learning algorithm with pruning for feature selection produced mixed results. On one hand, the pruning removed most of the new directives that are not included in the original set and the directives removed in the second experiment. It also left most of the directives not pruned in the pruning experiment. On the other hand, the results of this experiment are worse than the original pruning algorithm. We believe that the “new” directives affect the ability of the structure learning algorithms to capture the relations between the ’good’ directives and the coverage attributes. Therefore, to improve the results of this experiment, we will try to learn the structure of the network again after the pruning.

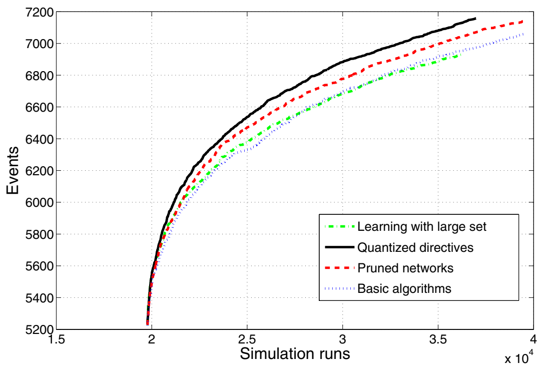


Fig. 5. Coverage progress of aggregated results of all experiments

6 Conclusions and Future Work

Closing the loop from coverage analysis to directives to the stimuli generation and reaching high coverage is one of the most difficult and time consuming challenges a verification team must face. In this paper we presented an automatic technique for constructing data-driven CDG engine based on Bayesian networks that is aimed at providing coverage boosting with minimal human effort.

Early results, obtained on a coverage model used in the unit verification of an IBM mainframe processor, indicate that our technique is able to achieve the coverage booster approach. Namely, improve coverage in a significant way with minimal human effort both in the construction and activation of the CDG system. Still, there is a lot of room for improvements and issues that we need to address. First is the need in an automatic process to determine the quality of subsets of directives in order to choose high quality subsets. Another issue is automatic construction of networks with soft evidence. We are currently working on these issues and we believe that improving these weaknesses will results with higher quality booster.

It is clear that the extreme goal, the Holy Grail, is to realize full coverage closure in a totally automatic way without any human effort even to set-up the automatic scheme. This goal was not yet achieved, and we assume that its complexity might cause it to stay an eluding one for the foreseeable future. Yet, we believe that the coverage booster approach has more potential of getting us closer to this Holy Grail and provide more benefits to the verification process.

References

1. Wile, B., Goss, J.C., Roesner, W.: Comprehensive Functional Verification -The Complete Industry Cycle. Elsevier, Amsterdam (2005)
2. Piziali, A.: Functional Verification Coverage Measurement and Analysis. Springer, Heidelberg (2004)

3. Fine, S., Ziv, A.: Coverage directed test generation for functional verification using Bayesian networks. In: Proceedings of the 40th Design Automation Conference, pp. 286–291 (2003)
4. Fournier, L., Ziv, A.: Using virtual coverage to hit hard-to-reach events. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 104–119. Springer, Heidelberg (2008)
5. Wagner, I., Bertacco, V., Austin, T.: Microprocessor verification via feedback-adjusted Markov models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(6), 1126–1138 (2007)
6. Bose, M., Shin, J., Rudnick, E.M., Dukes, T., Abadir, M.: A genetic approach to automatic bias generation for biased random instruction generation. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001, pp. 442–448 (2001)
7. Hsiou-Wen, H., Eder, K.: Test directive generation for functional coverage closure using inductive logic programming. In: Proceedings of the High-Level Design Validation and Test Workshop, pp. 11–18 (2006)
8. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference. Morgan Kaufmann, San Francisco (1988)
9. Fine, S., Ziv, A.: Enhancing the control and efficiency of the covering process. In: Proceedings of the High-Level Design Validation and Test Workshop, pp. 96–101 (2003)
10. Fine, S., Freund, A., Jaeger, I., Mansour, Y., Naveh, Y., Ziv, A.: Harnessing machine learning to improve the success rate of stimuli generation. IEEE Transactions on Computers 55(11), 1344–1355 (2006)
11. Cooper, G.F., Herskovits, E.: A Bayesian method for the induction of probabilistic networks from data. Journal of Machine Learning 9(4), 309–347 (1992)
12. Laskey, K.B., Myers, J.W.: Population markov chain monte carlo. Journal of Machine Learning 50(1-2), 175–196 (2003)
13. Chickering, D.: Optimal structure identification with greedy search. Journal of Machine Learning Research 3, 507–554 (2002)
14. Friedman, N.: The Bayesian structural EM algorithm. In: Proc. 14th Conf. on Uncertainty in Artificial Intelligence, pp. 129–138 (1998)
15. Ur, S., Yadin, Y.: Micro-architecture coverage directed generation of test programs. In: Proceedings of the 36th Design Automation Conference, pp. 175–180 (1999)
16. Rusakov, D., Geiger, D.: Asymptotic model selection for naive Bayesian networks. J. Mach. Learn. Res. 6, 1–35 (2005)
17. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. 3, 1157–1182 (2003)
18. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artif. Intell. 97(1-2), 273–324 (1997)
19. Cover, T.M., Thomas, J.A.: Elements of Information Theory. John Wiley, Chichester (1991)
20. Heckerman, D.: A tutorial on learning with Bayesian networks. Technical report, Microsoft Research, Redmond, Washington (1996)

Efficient Decision Procedure for Bounded Integer Non-linear Operations Using SMT($\mathcal{LIA}$)

Malay K. Ganai

NEC Labs America, Princeton, NJ, USA

Abstract. For the verification of complex designs, one often needs to solve decision problems containing integer non-linear constraints. Due to the undecidability of the problem, one usually considers bounded integers and then either linearizes the problem into a SMT($\mathcal{LIA}$) problem (i.e., the theory of linear integer arithmetic with Boolean constraints) or bit-blasts into a SAT problem. We present a novel way of linearizing those constraints, and then show how the proposed encoding to a SMT($\mathcal{LIA}$) problem can be integrated into an incremental lazy bounding and refinement procedure (LBR) that leverages on the success of the state-of-the-art SMT($\mathcal{LIA}$) solvers. The most important feature of our LBR procedure is that the formula need not be re-encoded at every step of the procedure but rather, only bounds on variables need to be asserted/retracted, which are very efficiently supported by the recent SMT($\mathcal{LIA}$) solvers. In a series of controlled experiments, we show the effectiveness of our linearization encoding and LBR procedure in reducing the SMT solve time. We observe similar effectiveness of LBR procedure when used in a software verification framework applied on industry benchmarks.

1 Introduction

For the theory of integer non-linear operations, the decision problem is un-decidable. Therefore, the decision procedures for such a theory typically assume bounded integer operands. Such an assumption is generally justified, given the verification problems arising from various hardware/software domains use finite width integer (words). Such non-linear operations do arise often, though used sparingly, in system design and verification. Traditionally, integer non-linear operations are handled in Boolean logic by bit-blasting all operands. However, there are some inherent disadvantages in reasoning at the Boolean level. Propositional translations of richer data types such as integers, and high-level expressions such as arithmetic, lead to large bit-blasted formulas. Moreover, the high-level semantics such as arithmetic are often "lost" in such a low-level translation, thereby, the SAT search becomes more difficult [1].

With the growing use of high-level design abstraction to capture today's complex design features, the focus of verification techniques [2] has been shifting from propositional reasoning towards Satisfiability Modulo Theory (SMT) solvers [3,4], and SMT-based verification methods such as bounded model checking (BMC) [5]. To capitalize on these workhorses, encoding for integer non-linear operations such as multiplication can be carried out using linearization, i.e., one of the operands of multiplication is bit-blasted, and the result is expressed as linear arithmetic operations.

2 Our Approach: Overview

In our effort to build an efficient decision procedure for bounded integer non-linear operations using SMT($\mathcal{LIA}$), we propose the following, as illustrated in the overall flow of our approach in Figure 1.

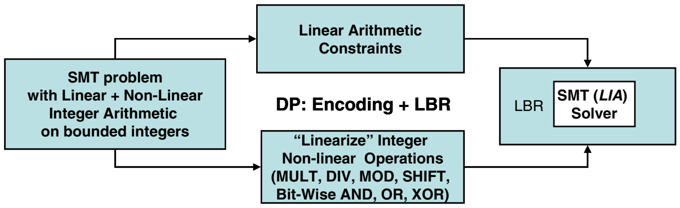

SMT problem with Linear + Non-Linear Integer Arithmetic on bounded integers
Linear Arithmetic Constraints
DP: Encoding + LBR
"Linearize" Integer Non-linear Operations (MULT, DIV, MOD, SHIFT, Bit-Wise AND, OR, XOR)
LBR
SMT (*LIA*) Solver

Fig. 1. Decision procedure for mixed integer linear and non-linear operations

First, we propose a novel and an improved linearization encoding of bounded integer non-linear arithmetic operations such as *multiplication*, *divide*, *mod*, and bitwise operations such as *shift*, *or*, *xor*, and *and*. This encoding translates non-linear operations *exactly* into SMT($\mathcal{LIA}$), and extends the reasoning power seamlessly to decision problems with mixed linear and non-linear integer operations without the need to bit-blasting all operands. The encoding, based on bitwise structural decomposition, is specifically geared towards improving the search of SMT($\mathcal{LIA}$) solver.

Second, we propose an incremental lazy bounding and refinement (LBR) decision procedure on the encoded SMT($\mathcal{LIA}$) problem to leverage on the success of the state-of-the-art SMT($\mathcal{LIA}$) solvers [3]. Our LBR procedure incrementally formulates partially constraint problems in successive iterations. In each iteration, the bounding constraints are either *tightened* or *relaxed*, depending on whether the result was spuriously satisfiable (due to under-constraining) or spuriously unsatisfiable (due to over-constraining), respectively, in the previous iteration. Such a procedure has several advantages: (i) it avoids re-encoding of the formula, thereby, allows the SMT($\mathcal{LIA}$) solver to capitalize on the learnt facts and pruning of the search space so far; (ii) it exploits the capability of the recent SMT($\mathcal{LIA}$) solvers [3] to assert/retract constraints efficiently through their use of a fixed tableau; and (iii) it effectively guides the SMT($\mathcal{LIA}$) solver to a faster resolution, when combined with our linearization encoding. Third, we propose linearization criteria for choosing non-linear operands for Booleanization. Such selection criteria are geared towards reduction of the size of the encoded problem and reduction on the number of iterations of LBR procedure, thereby, minimize the inherent linearization overhead. In a series of controlled experiments, we observe that our linearization encoding helps reduce the SMT solve time significantly compared to a previous linearization approach [6]. Further, our LBR procedure integrated with the improved linearization is quite effective against the state-of-the-art SMT($\mathcal{BV}$) solver [7]. To evaluate further on verification benchmarks, we integrated our decision procedure in a SMT-based BMC tool [5]. This tool serves as a backend engine for software verification framework called F-SOFT [8], targeted for finding standard programming bugs in embedded system software written in C. Verification is performed via a translation of the given C program to a finite state circuit model. Here we target typical programs that use linear operations more often than non-linear operations. Our verification experimentation show the effectiveness of our decision procedure.

3 Related Work

Linearization for non-linear datapaths has been studied in the context of RTL verification [9, 10, 11]. In these approaches, linear arithmetic constraints are generated for linear and non-linear datapaths, and are encoded into integer linear programming (ILP) expressions. In [10], a special attention was given to the modulo semantics. In [6], a linearization encoding with Booleanization (bit-extraction) was used to generate SMT($\mathcal{LIA}$), i.e., the theory of Linear Integer Arithmetic with Boolean constraints. In these approaches, integer bounds were added *eagerly* as bounding constraints. To handle modulo semantics, additional constraints were also added *eagerly*.

As reconfirmed in our experiments, the integer bounding constraints cause the solver to slow down significantly, especially, when added eagerly. To overcome this, approaches [12, 13] use abstraction/refinement of bounds in decision procedure for solving bit-vector and Presburger theories. Other approaches [14, 15] use un-interpreted functions for abstracting datapaths, accompanied with iterative refinement steps. These approaches are based on a bit-blasted encoding, and therefore, they make it difficult to refine the formula without re-encoding it. Re-encoding in Boolean domain typically "destroys" the learning done by DPLL-style SAT solvers [16,17,18] and thereby affects the performance of the solvers, as the learning need to be rediscovered. Moreover, it is not obvious how to guide SAT solvers using high-level constraints, as SAT solvers are usually "oblivious" of arithmetic expressions.

The theory of bit-vector SMT($\mathcal{BV}$) is inherently non-linear. In practice, SMT($\mathcal{BV}$) solvers use SMT($\mathcal{LIA}$) in the last stage after linearizing bit-vector operations [19, 10] or use complete bit-blasting after applying re-write and inference rules at the preprocess/online stage [3, 20, 21, 13, 22, 7]. In [23], a decision procedure for non-linear congruences (i.e., equalities on bounded integers) is presented, however, it does not address non-linear inequalities.

Computer algebra systems such as Maxima [24] are intended for the manipulation of symbolic and numerical expressions including factorization, and solving linear equations. However, such systems can not be directly applied to verification methods such as SMT-based BMC.

Outline. The rest of the paper is organized as follows: In Section 4, we give the basic notations and some background on SMT, and modeling C programs and data overflow; in Section 5, we present our linearization encoding; in Section 6, we discuss our LBR procedure; in Section 7, we present our linearization criteria; in Section 8, we evaluate and compare our approach in a series of experimentation; and in Section 9, we summarize our work with concluding remarks.

4 Background

4.1 Mixed Expressions and SMT

In this paper, we consider decision problems with Boolean expressions *bool-expr* and integer term expressions *term-expr*. Boolean expressions include Boolean operations such as "and" ($\wedge$), "or" ($\vee$), and "not" ($\neg$) on *bool-var* (Boolean variables) or *bool-expr*. Integer term expressions comprise linear operations such as addition ($+$), subtraction ($-$), constant multiplication ($*_c$); non-linear operations such as multiplication ($*$), divide ($/$), and modulo ($\%$); and bitwise operations such as left shift ($<<$), right shift ($>>$), bit-wise and ($\&$), bit-wise or ($||$) and bit-wise xor ($\otimes$) on *integer-var* (integer variables) or *term-expr*. We use ITE(*bool-expr*, *term-expr*, *term-expr*) as a shorthand to denote the if-then-else operator.

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to a background decidable first-order equational theory $\mathcal{T}$ (SMT($\mathcal{T}$)). In this work, we focus on deciding quantifier-free formulas (mixed expressions) by translating *term-expr* to the theory of linear arithmetic over integers ($\mathcal{LIA}$). We define $\mathcal{LIA}$-var as a theory variable interpreted in the domain of $\mathcal{LIA}$, and $\mathcal{LIA}$-term as a first-order term build from linear operators $(+, -, *_c)$ and $\mathcal{LIA}$-var. Such a theory constitutes a conjunction of integer linear arithmetic constraints (LAC) $\sum_i a_i *_c x_i \leq d$. The problem of checking the decidability of LAC over integer domain is NP-complete. SMT($\mathcal{LIA}$) is the problem of deciding satisfiability of Boolean expressions obtained on applying Boolean connectives on propositional atoms and relational operators $(==, \leq)$ on two $\mathcal{LIA}$-terms.

In a DPLL-based SMT($\mathcal{LIA}$) solver [4,3], SMT($\mathcal{LIA}$) formula is solved by combination of Boolean SAT and $\mathcal{LIA}$-solver, wherein $\mathcal{LIA}$-solver solves each successive integer LAC problem generated. Typically, an $\mathcal{LIA}$-solver employs a layered approach wherein an integer LAC problem is first approximated into a real LAC problem which then is solved using an incremental simplex-based $\mathcal{LRA}$-solver. A *branch and cut* strategy is used to overcome the incompleteness of the $\mathcal{LRA}$-solver. (Note, the theory of $\mathcal{LRA}$ constitutes a conjunction of real LAC.)

As successive LAC problems share subproblems, there has been a lot of research to devise incremental solvers to exploit the sharing. Some of the Simplex-based [25] methods [26] use incremental versions; a *tableau* is constructed where rows (corresponding to LAC) are added and deleted incrementally during DPLL search. It was shown [3] that such incremental updates of the tableau are expensive. To reduce the cost, the solver [3] operates on a transformed problem where the tableau is fixed (i.e., rows are not added/deleted) during the decision process, but constraints corresponding to variable-bounds change during the search. A SMT($\mathcal{LRA}$) formula ϕ is first transformed into an equi-satisfiable formula $\phi_{eqn} \wedge \phi_{pred}$, where the formula ϕ_{eqn} represents the conjunction of linear equations $Ax = 0$, and ϕ_{pred} represents the Boolean combination of predicates of the form $x \bowtie b$ where $\bowtie \in \{==, <, >, \leq, \geq\}$ and b is an integer constant. *Note, that the matrix A is fixed during the decision process. It is the set of inequalities of the form $l_i \leq x_i \leq u_i$ that changes during the decision process where l_i, u_i correspond to the bounds of each variable x_i.* These inequalities can be asserted/retracted efficiently without undoing the previous pivoting operations. We exploit this incremental capability in our decision procedure.

4.2 Booleanization

A bounded integer variable $x \in [0, 2^N)$ can be related with its Boolean decomposition variables $B_{N-1}B_{N-2} \ldots B_0$ using the following linear and Boolean constraints, respectively, where x_k is an *integer-var* such that $x_k \in [0, 1]$.

$$x = \sum_{k=0}^{N-1} 2^k *_c x_k \tag{1}$$

$$\bigwedge_k B_k = (x_k == 1) \tag{2}$$

We refer to such a bit-extraction approach as *Bit-wise Relational Decomposition* (BRD), as used in the previous linearization [6]. Later, we contrast it with our structural decomposition where we relate B_k with x directly using arithmetic expressions.

4.3 Modeling C Program and Data Overflow

Using a software verification framework F-SOFT [8], we build a finite model from a given C program using Boolean and arithmetic expressions, derived automatically by considering the control and data flow of a program under the assumptions of *bounded data* and *bounded recursion*.

In C, the integer terms are either *unsigned* or *signed*. For N-bit arithmetic operations, each *signed* integer x has interval bound $[-2^{N-1}, 2^{N-1})$, i.e. -$2^{N-1} \leq x < 2^{N-1}$, and *unsigned* term integer term u has interval bound $[0, 2^N)$ i.e., as $0 \leq u < 2^N$. As per the C/C++ language standards (C99/C++98), signed overflow is *unspecified* and unsigned overflow is reduced to modulo 2^N, i.e., it "wraps around." (Similarly, for underflow). An implementation can choose to overflow signed integers using modulo operation (C99: H.2.2-clause); however, we are considering verification of implementation independent C-programs where programmers cannot assume any particular handling of signed overflow. We focus on handling signed/unsigned bounded integers as per the standard, wherein, we either detect or do not allow overflow in signed integers. In contrast, typically a $SMT(\mathcal{BV})$ solver would handle a signed (or unsigned) overflow in bit-vector arithmetic implicitly, as "wrap-around". By encoding signed/unsigned integers differently, we handle overflows in C programs as per C99. (Interested readers may check out [27] for encoding of casting and overflows).

5 Encoding Non-linear Arithmetic Operations

In the following, for simplicity, we assume all integers are *signed* with a system interval bound $[-2^N, 2^N)$. We assume that overflow/underflows are handled as per C99/C++98 standard.

5.1 Bitwise Structural Decomposition (BSD)

To relate a bounded *integer-var* $u \in [0, 2^N)$ with its Boolean decomposition variables $B_{N-1}B_{N-2}\ldots B_0$, we define new Boolean predicates B_k and integer terms t_k for $k = N$-1 to 0 in a mutually recursive manner as follows:

$$B_k = (t_k \geq 2^k) \;\; (0 \leq k \leq N-1) \tag{3}$$

$$t_{k-1} = ITE(B_k, t_k - 2^k, t_k) \;\; (1 \leq k \leq N-1) \tag{4}$$

$$t_{N-1} = u \tag{5}$$

Note that we do not introduce any new *integer-var* but do introduce additional N-1 linear constraints, as compared to the BRD approach (Eqn 1-2) [6]. *Example:* Let $u = 13$. We obtain $t_3 = 13$, $B_3 = (13 \geq 2^3) = 1$; $t_2 = 13 - 2^3 = 5$, $B_2 = (5 \geq 2^2) = 1$; $t_1 = 5 - 2^2 = 1$, $B_1 = (1 \geq 2^1) = 0$; $t_0 = t_1 = 1$, $B_0 = (1 \geq 2^0)$. Note, $B_3B_2B_1B_0 = 1101$ is a bit-representation of $u = 13$.

The BRD approach relies on the solver to guess the variables B_k, whereas in our BSD approach, the structural relation between the Boolean and intermediate terms expressions are explicitly captured. Intuitively, such constraints provide "pre-computed learning" to a search process; which, otherwise, have to be learnt (e.g., in BRD), incurring additional backtracking cost. Our experimental results confirm this observation.

5.2 Integer Multiplication (MULT)

Let $z = u * v$ assuming $u, v \geq 0$. Assume, the operand u is *chosen* for bit-extraction. Let $U_{N-1}U_{N-2} \ldots U_0$ denote the bitwise decomposition of the operand u as obtained using either Eqn 1-2 or Eqn 3-5. For $k = 0$ to $N - 1$, with $r_0 = 0$, we define partial sums as follows:

$$r_{k+1} = r_k + ITE(U_k, 2^k *_c v, 0) \tag{6}$$

Finally, we relate z with r_N, i.e., $z = r_N$. We use $*_L$ to denote the above linearization of multiplication, and henceforth, refer to it as *linearized multiplier*. Note, it is easy to verify that $u * v = u *_L v$ holds if $(u < 2^N)$. Now, we focus on the general case $z = x * y$, where the operands x, y can take negative values. We define $u = ITE(x < 0, -x, x)$ and $v = ITE(y < 0, -y, y)$. We encode $w = u * v$ as described above. We relate z with w as $z = ITE(y < 0 \otimes x < 0, -w, w)$ where $\otimes$ denote the Boolean XOR operation. Note, we encode $y = ITE(S, a, b)$ into SMT($\mathcal{LIA}$) by adding two CNF-clauses $(!p + A)(p + B)$ with predicates $A = (y == a)$, $B = (y == b)$.

For a given set of non-linear multiplier operations, ideally we would like to have a fewer such u operand chosen for bit-extraction to reduce the overall size of the encoded problem after linearization. Further, a choice of operand for bit-extraction also influence the LBR decision procedure (details of selection criteria in Section 7).

5.3 Divide (DIV)

For an ease of description, we assume integers are positive. We add additional constraints as described above for a general case. Let $w = u/v$, with $u \geq 0, v > 0$. We introduce an *integer-var* term t and add following bounding constraints:

$$u - v + 1 \leq t * v \leq u \tag{7}$$

Note that for $u, v > 0$, t equals $\lfloor u/v \rfloor$. When v is a constant, we use a constant multiplier $*_c$; otherwise, we use a linearized multiplier $*_L$. To handle the trivial cases, we express the divide result w as follows:

$$w = ITE(u < v, 0, ITE(u = v, 1, ITE(v = 1, u, t))) \tag{8}$$

5.4 Modulo (MOD)

Let $w = u\%v$, where $u \geq 0, v > 0$. Similar to DIV, we introduce an *integer-var* term t and add bounding constraints on t as in Eqn 7. To handle the trivial cases, we express the modulo result w as follows:

$$w = ITE(u < v, u, ITE(u = v, 0, ITE(v = 1, 0, u - t))) \tag{9}$$

5.5 SHIFT

Left and right shifts are expressed in terms of constant divide and multiplier, respectively. Let $v_r = u >> w$, $v_l = u << w$ where $u, w \geq 0$. We relate v_r and v_l as follows:

$$v_r = ITE(w = 0, u, \ldots ITE(w = N, u/2^N, 0) \ldots) \tag{10}$$

$$v_l = ITE(w = 0, u, \ldots ITE(w = N, w *_c 2^N, 0) \ldots) \tag{11}$$

Note, number of ITE is equal to N as each operand has bound $[-2^N, 2^N)$.

5.6 Bit-Wise Operations

Let $z = u \; bop \; v$ assuming $u, v \geq 0$, and $bop \in \{\&, \|, \otimes\}$. Let $U_{N-1}U_{N-2} \ldots U_0$ and $V_{N-1}V_{N-2} \ldots V_0$ denote bitwise decomposition of the operands u and v, respectively. For $k = 0$ to $N - 1$, with $r_0 = 0$, we define the partial sum as follows:

$$r_{k+1} = r_k + ITE(U_k \; bop \; V_k, 2^k, 0) \tag{12}$$

Finally, we relate z with r_N, i.e., $z = r_N$.

6 Lazy Bounding and Refinement Procedure (LBR)

We discuss our LBR decision procedure that incrementally constraints the formula with bound constraints, and avoids re-encoding the formula. Before we present the procedure, we describe various notations and design choices employed.

Let ϕ be an encoded SMT($\mathcal{LIA}$) formula obtained after linearization of non-linear expressions as described in the previous section. To minimize the inherent linearization overhead, we selectively *choose* to bit-extract one non-linear operand of MULT over the other, based on *Linearization Criteria* (LC) as described in Section 7. Using the criteria, we partition the terms in ϕ into exclusive sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, where $x_i (\in X)$ represents a term *not chosen* for bit-extraction (referred as X term) and $y_i (\in Y)$ represents a term *chosen* for bit-extraction (referred as Y term), respectively. Note, the sub-terms in Eqn 6 belong to X set. For ease of presentation, we assume that all variables $v \in X \cup Y$ are signed and are bounded by system interval bound $[-2^N, 2^N)$, i.e., $-2^N \leq v < 2^N$. One can use interval analysis (such as [28]) statically on the program to obtain a conservative but much tighter interval bound than system interval bound $[-2^N, 2^N)$. In the following, we consider bounding constraints of the form $-b(v) \leq v < b(v)$ where $v \in X \cup Y$ and $b(v)$ represents the current bound size for the variable v. Note, $b(v) \leq 2^N$.

In the presence of non-linear operations, we are required to bound Y terms in the formula (as noted earlier, $x*y = x*_L y$ if $y \in Y$ and $b(y) \leq 2^N$). Starting with a smaller bound $[-2^{b(y)}, 2^{b(y)})$ $(b(y) < N)$ for Y terms, predicates $B_k = (t_k \geq 2^k)$ (Eqn 3) can be simplified to `false` for $k \geq b(y)$; which in turn, can lead to simplification of the overall-formula. Therefore, we choose Y terms for bound-relaxing, as further explained with an example. Consider the problem of *finding* x_1, y_1 *such that* $(x_1 *_L y_1 = 3*2039)$. Note, with $y_1 = 3$, we can solve the problem. Clearly it would be desirable to start with a small bound and relax it iteratively.

To exploit incremental solving capability of the recent SMT($\mathcal{LIA}$) solvers, we *do not want to re-encode* the formula when some bounds change. Re-encoding of the formula loses all the information that was learnt in the previous run. To achieve our goal, we decide to bit-extract all the non-linear operands (from Y set) using the system bound, i.e., $[-2^N, 2^N)$ *once*, but *assert* (i.e., add)/ *retract* (i.e., remove) the bounding constraints on $y \in Y$ terms *selectively* and *incrementally*. Using this incremental formulation, we *relax* the bounds of $y \in Y$ terms when the formula is unsatisfiable due to insufficient bound size $b(y)$.

In our experiments, we also found that adding the bounding constraints *eagerly*, i.e., $-2^N \leq x < 2^N$ for all $x \in X$ *also* increases solve time. Therefore, we add the bounding constraints *lazily*, i.e., we *tighten* bounds only when a model returned does not meet the bound constraints $[-2^N, 2^N)$ on one or more X terms. Many times, as observed in our SMT-based BMC verification, BMC instances can be shown unsatisfiable quickly without bounding all X terms. In short, we relax/tighten bounding constraints on Y/X terms lazily. Intuitively, interchanging X/Y terms for tightening/relaxing would worsen the performance.

We present our LBR[1], as shown in Algorithm 1.

Algorithm 1. Lazy Bounding and Refinement Algorithm

```
1: input: SMT($\mathcal{LIA}$) formula $\phi$ without bounding constraints for non bit-extracted terms $X = \{x_1, \ldots, x_n\}$ and bit-extracted terms set $Y = \{y_1, \ldots, y_m\}$ s.t. $X \cap Y = \emptyset$
2: output: $SAT/UNSAT$
3:
4: SMT_Init($\phi$) {Encode the (unbounded) logical formula}
5: $X_b := \emptyset$ {Set of $x \in X$ with $[-2^N, 2^N)$ interval bound}
6: for all $y \in Y$ do
7:     $b(y) := 2$ {Initial bound size for $y$}
8:     SMT_Assert($-b(y) \leq y < b(y)$) {Bound $y$ variables}
9: end for
10: {Iterate through tightening/relaxing}
11: loop
12:     $is\_sat$ := SMT_Check() {Invoke SMT($\mathcal{LIA}$) solver}
13:     if ($is\_sat$ = $true$) then
14:         Let $\alpha(x)$ be the satisfying assignment for each variable $x \in X$
15:         $X'_b := \emptyset$
16:         for all $x \in X \backslash X_b$ do
17:             if $\neg(-2^N \leq \alpha(x) < 2^N)$ then
18:                 {model value does not satisfy bound}
19:                 SMT_Assert($-2^N \leq x < 2^N$) {Bound $x$ variables}
20:                 $X'_b := X'_b \cup \{x\}$ {New set of bounded $x$ variables}
21:             end if
22:         end for
23:         if ($X'_b = \emptyset$) then
24:             {all model values satisfying}
25:             return $SAT$ {Satisfiable}
26:         end if
27:         $X_b := X_b \cup X'_b$ {update $X_b$}
28:     else
29:         {UNSAT result}
30:         {Check which bound constraints $-b(y) \leq y < b(y)$ failed}
31:         Let $Y_b := \{y \mid$ Either $(-b(y) \leq y)$ OR $(y < b(y))$ cause for UNSAT, and $b(y) < 2^N\}$

32:         if ($Y_b = \emptyset$) then
33:             return $UNSAT$ {Unsatisfiable}
34:         end if
35:         for all $y \in Y_b$ do
36:             SMT_Retract($-b(y) \leq y < b(y)$) {Retract the tight bounds}
37:             $b(y) := 2 * b(y)$ {Relax by factor of 2}
38:             SMT_Assert($-b(y) \leq y < b(y)$) {Assert the relaxed bounds}
39:         end for
40:     end if
41: end loop
```

[1] The main difference from [13] are in *incremental* and *SMT* formulation. In LBR, we start with an under-constrained formula, whereas in [13] every term is explicitly bounded. Unlike [13], we do not need to analyze unsat core.

LBR takes a formula ϕ with partitioned sets X and Y, system bound $[-2^N, 2^N)$ and outputs $UNSAT$, or SAT with a satisfying model. LBR invokes a SMT($\mathcal{LIA}$) solver, iteratively without re-encoding the formula ϕ, and incrementally asserting and retracting bounding constraints. Following procedures are typically supported in such a solver: `SMT_Init` to add initial set of logical expressions, `SMT_Assert` to assert constraints, `SMT_Retract` to retract previously added constraints, and `SMT_Check` to check for satisfiability.

Let X_b denote the subset of X terms that currently have bounding constraints. Initially, LBR starts with no bounding constraints on X terms, i.e. $X_b = \emptyset$, but bounding constraints on all Y terms corresponding to the bound size $b(y) = 2$. These constraints together with the formula ϕ result in a partially constrained formula that is checked for satisfiability. If the formula is satisfiable, the satisfying values $\alpha(x)$ of $x \in X \backslash X_b$ terms is checked against the system bounds. Those variables violating the bound constraints are *tightened* by asserting the corresponding constraints $-2^N \leq x < 2^N$. The set X_b is also updated accordingly. On the other hand, if all values of x are within the system bound, i.e., the assignment is not spurious, SAT is returned.

If the formula is unsatisfiable, we need to find out if the unsatisfiability is due to insufficient bound sizes $b(y)$ of $y(\in Y)$ terms [2]. We obtain the unsatisfying assertions, i.e., the bounding constraints either of the form $-b(y) \leq y$ or $y < b(y)$ that were sufficient to cause unsatisfiability. From these, we construct a set Y_b such that $b(y) \neq 2^N$. If the set Y_b is empty, then clearly the formula is unsatisfiable, and UNSAT is returned; otherwise, for each $y \in Y_b$, we first retract the corresponding (tight) bound constraints, relax them by a factor 2, and then assert them back. Note, the initial bound $b(y)$ and the scale factor can be chosen heuristically. We continue with the next iteration.

Theorem 1. *The algorithm* LBR *decides correctly and always terminates.*

Proof. Please refer [27].

7 Linearization Criteria

The size of the encoded problem after linearization depends directly on the number of Y terms (which are chosen for bit-extraction). Further, the number of iterations in the `LBR` procedure also depends on the choice of Y terms. To reduce the size of encoded problem and number of iterations in `LBR`, and thereby, minimize the inherent linearization overhead, we use the following criteria to selectively *choose* to bit-extract the non-linear operand u of MULT $z = u * v$ over the other operand v while encoding $z = u * v$:

- *Rule 1 (R1): The operand u is an input to more MULT compared to v.* The goal is to reduce the number of operands considered for bitwise decomposition in a given non-linear decision problem, thereby reducing linearization constraints. In general, one can formulate it as a *vertex cover* problem [29].
- *Rule 2 (R2): The operand u has a fewer MULT in its transitive input compared to v.* Intuitively, we do not want to bound a term when it can be implied by the bounding constraints imposed on its transitive inputs. Such a rule is intended to reduce the number of bound relaxing iterations (line 37, Algorithm 1).
- *Rule 3 (R3): The operand u has lower bound size compared to v.* Clearly, a lower bound size on u would require fewer bound relaxing steps in the LBR procedure.

[2] Finding minimal constraints might be as hard as generating unsat core (in terms of computation overhead), but one can use heuristics to obtain a non-minimal set quickly.

Example: Let there be three (non-linear) multipliers in a decision problem, i.e., $w_2 = w_1 * u_1$, $w_1 = u_2 * u_2$, and $w_3 = u_2 * u_3$. Potentially, there are four sets of operands for bit-extraction, i.e., $Y_a = \{u_1, u_2\}$, $Y_b = \{w_1, u_2\}$, $Y_c = \{u_1, u_2, u_3\}$, and $Y_d = \{w_1, u_2, u_3\}$. Using the rule $R1$, we would prefer a set Y_a or Y_b over the remaining sets. Intuitively, a smaller set reduces linearization constraints, and hence size of the encoded problem. Among Y_a and Y_b, we choose Y_a using the rule $R2$. The iterative bounding constraints on u_2 would impose bounding constraints on w_1 implicitly. Thus, we would rather not choose w_1 for bit-extraction, as adding explicit bounding constraints on w_1 (line 12, Algorithm 1) may over-constrain the formula, resulting in more iterations.

8 Experiments

We implemented our LBR algorithm over the SMT solver *yices-1.0.11* [30]. Our experiments were conducted on a single threaded environment, on a workstation with 3.4GHz, 2GB of RAM running Linux. We conducted three sets of experiments.

In the first and second sets, we carried out a series of controlled experimentation, where we compared a) various lazy and eager combination of X and Y term sets, b) bitwise decomposition in our linearization against a previous approach [6], c) effectiveness of linearization criteria, with and without, and d) our decision procedure against a state-of-the-art SMT solver Z3 (stable version 1.1) [31] supporting non-linear in SMT($\mathcal{BV}$). (Note, we compare our method against only those approaches that are widely used in verification application, and did not compare against algebra system [24] geared towards the manipulation of numerical expressions.)

For the third set, we integrated our LBR decision procedure as the backend solver in SMT-based BMC [5] in software verification framework F-SOFT [8]. We used this framework to check reachability properties in a set of simple and realistic software programs, respectively.

8.1 Controlled Experimentation

We evaluated and compared various approaches on non-linear decision problems arising from combination of parameterized multivariate polynomials of various degrees. For this experiment, we used a timeout of 600s for each run. As Z3 library is available on Windows platform only, we gave extra time, i.e., a limit of 1000s to Z3 solver (for fair comparison).

We used system interval bound $[-2^N, 2^N)$ with $N = 31$, where all integer variables are signed.

Comparing Decision Procedure LBR. We compare LBR procedure against two other variations depending on whether the bounding constraints (BC) for X and Y variables are added *eagerly* or *lazily*.

– LBR-LE Lazy BC on X, but Eager BC on Y
– LBR-EL Eager BC on X, but Lazy BC on Y
– LBR-LL Lazy BC on both X and Y

In all three algorithmic variations, we used linearization encoding based on our bitwise structural decomposition (Eqn 3-5). Further, we partition the terms into X and Y sets using the linearization criteria discussed before. Note that for the same linearization criteria, the number of iterations needed for bound relaxing (i.e., on Y terms) are also the same.

Current version of *yices-1.0.11* do not provide API to obtain the set of unsatisfying bounding constraints when the result is UNSAT, i.e., which assertions failed. Therefore, for this experimentation, we used a simple heuristic where we construct the set Y_b with *all* y such that $b(y) \neq 2^N$. Since this heuristic is sub-optimal, our results could be improved further.

Comparing Booleanization. To compare our Booleanization against the previous approach [6], we used the bitwise relational decomposition (Eqn 1-2) in linearization, and combined with the LBR algorithm with lazy bounding constraints on both X and Y terms. We call this approach BRD. This approach also uses linearization criteria to partition terms into X and Y sets. Note, approach LBR-LL compares fairly with BRD, as they differ only in the Booleanization methods used.

Comparing Linearization Criteria. To show the effectiveness of linearization criteria, we used opposite criteria that do not satisfy one or more rules $R1 - R3$ to partition the terms into X and Y sets. We used these partition sets in our linearization, and combine it with LBR algorithm where bounding constraints on both X and Y terms are added lazily. We refer this approach as NLC. This approach uses the bitwise structural decomposition (Eqn 3-5). Note, approach LBR-LL compares fairly with NLC, as they differ only in the use/no use of linearization criteria, respectively.

Comparing LBR with Z3. We also compare our approach with Z3 SMT solver [31] where we used system bound $[-2^N, 2^N)$ to encode bit-vector formula. We also added constraints to prevent overflow.

Benchmarks. We picked a handful of parameterized multivariate polynomials with degrees 2 and 3, and selected the parameters from a set of product of prime numbers. These polynomials, with their integer roots, are shown below.

$f_1(x, y) = x^2y + 11x^2 + 13xy + 143x + 30y + 330$
(roots: $x = -3, -10, y = -11$)
$f_2(x) = x^3 + 9x^2 + 27x + 27$ (root: $x = -3$)
$f_3(x, y) = 12xyz + 8xz + 36yz + 24z + 6xy + 4x + 18y + 12$
(root: $x = -3$)
$f_4(y) = y^3 + 27y^2 + 243y + 729$ (root: $y = -9$)
$f_5(x) = x^2 - 3x + 2$ (roots: $x = 1, 2$)
$f_6(x, y, c) = xy - c$ (roots: factors of a constant c)

Note, we dropped the $*$ notation for better readability. Our goal is to find the integer roots of the polynomials in various combinations. We pose the problem as a decision problem to be solved in our experimental framework. We experimented with two varieties of decision problems. In one type, the syntactic variables of the polynomials are the same, and in the other type, they are different, as explained below.

Let $arg(f_i)$ denote the set of variables in polynomial f_i. For example, $arg(f_1) = \{x, y\}$. A decision problem dp denoted as $a : \{f_i, \ldots, f_j\}$ represents the formula $f_i = 0 \wedge \ldots \wedge f_j = 0$ where a is S or D depending on whether the syntactic variables of $f_i, \ldots, f_j$ are same or not, respectively. For example, $dp \equiv D : \{f_1, f_2\}$ denotes the formula $(f_1(x_1, y_1) = 0) \wedge (f_2(x_2) = 0)$ where x_1, x_2, and y_1 are free input variables. On the other hand, $dp \equiv S : \{f_1, f_2\}$ denotes the formula $(f_1(x, y) = 0) \wedge (f_2(x) = 0)$ where x and y are free input variables. For parametric function f_6, we use $f_6(z = c)$ to denote the polynomial $xy = c$. The constant c is determined by the product of prime numbers, where the prime number corresponds to the largest prime number strictly smaller than 2^k, with $0 < k < N$.

We chose the set of benchmarks for a few good reasons. First, the set provides mixed arithmetic operations with both linear and non-linear types. Second, the set is simple to understand, and yet not straightforward to solve. Third, we easily obtain many decision problems using various combinations of these polynomials.

Experimental Results: We present our results in Table 1. Column 1 reports the decision problem $dp \equiv a : \{f_i \dots f_j\}$; Columns 2-8 report comparison numbers using linearization criteria. Specifically, Columns 2 and 3 report the size of the sets X and Y, i.e. n_X and n_Y, respectively. Column 4 reports the number of iterations (#I) needed by `LBR-EL`, `LBR-LL` and `BRD` approaches. (For `LBR-LE`, the number of iteration needed is one. For the rest, only bound relaxing on Y terms are needed. As they use the same linearization criteria, the number of iterations are also the same). Columns 5-8 report solve times for `LBR-LE`, `LBR-EL`, `LBR-LL`, and `BRD`, respectively. Columns 9-12 report comparison numbers without the linearization criteria i.e., using `NLC` approach. Specifically, Columns 9 and 10 report the size of the sets X and Y, i.e. n_X and n_Y, respectively; Columns 11 and 12 report the number of iterations needed (#I) and solve times, respectively. Columns 13 show the time taken by `Z3` on bit-vector encoding of the decision problem dp.

Discussion: We notice that `LBR-LE` approach times out (TO) in all cases. Such a result is not surprising, and only reassures that current SMT solvers are not geared towards an encoding where the bound constraints on Y terms are added *eagerly*. Performance of `LBR-EL` and `LBR-LL` are comparable. We argue that these set of examples have comparatively a fewer X terms, and therefore, the overhead of *eager* bounding constraints on X terms was not significant. However, in later experiments 8.3, we observe that `LBR-LL` approach outperforms `LBR-EL` approach when X terms are many. Comparing `LBR-LL` and `BRD` approaches, we observe that our bitwise structural decomposition boosts performance over previous bitwise relational decomposition [6] almost always. Comparing `LBR-LL` and `NLC` approaches, we observe that the use of linearization criteria results in a smaller set Y (compare Columns 3 and 10) and reduces the number of iterations (compare Columns 4 and 11). These translate into improved performance of `LBR-LL` over `NLC` almost always. Comparing `LBR-LL` with `Z3`, we find that `LBR-LL` does much better overall. Note, `Z3` times out in most cases. Thus, by guiding the SMT solvers using proposed bitwise structural decomposition and bound refinements, we exploit the internal incremental capability of SMT solvers without compromising the learning from previous iterations.

8.2 Experimentation with GCD/LCM Programs

Using our LBR decision procedure we evaluated on parameterized C programs [27] computing GCD (Greatest Common Divisor) and LCM (Least Common Multiple) based on Euclidean algorithm that use non-linear operations such as *modulo* and *division*. For each of these programs, we use an inverse function to describe the negated safety property that requires a non-trivial state space search. We generate verification conditions similar to CBMC [32].

We use *gcd*-$p1$-$p2$ to denote the inverse function: "*Given two parameters* $p1$ *and* $p2$*, find an integer pair* x *and* y*, such that* $x + y = p1$ *and* $GCD(x, y) = p2$*?*". Similarly, we use lcm-$p3$-$p4$ to denote the inverse function: "*Given two parameters* $p3$ *and* $p4$*, find an integer pair* x *and* y*, such that* $x + y = p3$ *and* $LCM(x, y) = p4$*?*". We pose these parameterized problems as decision problems to be solved in our experimental framework.

We experimented on the Linux platform as before, with time limit of 600 sec. For Z3, we used timeout of 1000s on Windows platform. We present our results in Table [2], showing comparison of LBR-LE, LBR-EL LBR-LL, BRD, and Z3. Column 1 reports the parameterized decision problem. Columns 2-5 report the performance results for LBR-LE, LBR-EL, LBR-LL, and BRD, respectively. Column 6 reports the number of iterations needed for LBR-EL, LBR-LL, and BRD approaches. Note, LBR-LE requires one iteration. For the rest, only bound relaxing on Y terms are needed. As they also use the same linearization criteria, the number of iterations are also the same. Column 7 reports the performance results of Z3.

Discussion: Observe that LBR-EL and LBR-LL approaches are comparable, and LBR-LL performs better than LBR-LE and BRD. Such results re-assure the superiority of our linearization encoding for the other non-linear operations (such as division and modulo) as well, for software programs. Observe, Z3 takes more time to solve gcd-$p1$-$p2$ examples and times out in all lcm-$p3$-$p4$ examples.

Table 1. Evaluating Lazy Bounding and Refinement Procedure

DP	With Linearization Criteria (LC)							Without LC				Z3 [31]
				LBR-LE	LBR-EL	LBR-LL	BRD [6]				NLC	
$a:(f_i,\ldots,f_j)$	n_X	n_Y	$\#I$	X : L Y: E T(sec)	X : E Y: L T(sec)	X : L Y: L T(sec)	X : L Y: L T(sec)	n_X	n_Y	$\#I$	X : L Y: L T(sec)	T(sec)
$S:\{f_2,f_3\}$	50	2	2	TO	4.1	3.8	58.4	46	6	4	528.1	1
$S:\{f_1,f_2\}$	38	2	2	TO	2.1	1.4	54.5	37	3	4	10.1	175
$S:\{f_1,f_3\}$	49	3	2	TO	10.9	10.8	286.2	47	5	4	123.0	38
$S:\{f_2\}$	19	1	2	TO	0.7	0.5	1.6	18	2	4	3.6	195
$S:\{f_1\}$	20	2	2	TO	3.1	3.7	9.5	20	2	2	0.8	1
$D:\{f_2,f_3\}$	50	4	2	TO	16.0	32.9	58.2	48	6	-	TO	TO
$D:\{f_1,f_4,f_5\}$	59	5	4	TO	104.3	121.8	560.8	58	6	7	297.1	328
$D:\{f_4,f_6(c_1)\}$	34	2	8	TO	19.2	43.3	47.5	33	3	8	68.9	TO
$D:\{f_4,f_6(c_2)\}$	34	2	10	TO	86.3	118.4	154.0	33	3	10	169.0	TO
$D:\{f_4,f_6(c_3)\}$	34	2	6	TO	10.1	13.9	21.4	33	3	7	50.9	TO
$D:\{f_4,f_6(c_4)\}$	34	2	4	TO	3.3	3.6	9.2	33	3	7	61.9	TO
$D:\{f_4,f_6(c_5)\}$	34	2	6	TO	10.7	14.3	13.2	33	3	7	60.6	TO
$D:\{f_4,f_6(c_6)\}$	34	2	4	TO	3.3	3.6	8.0	33	3	7	60.3	TO

Note:
L: Lazy, E: Eager, TO: Time Out, -: Not Applicable, $n_X = |X|$, $n_Y = |Y|$, $\#I$: number of iterations of LBR.
Constants: $c_1 = 251*509$, $c_2 = 1021*2039$, $c_3 = 31*61*127$, $c_4 = 3*7*13$, $c_5 = 61*127$, $c_6 = 3*7*13*31$

8.3 Experimentation with Benchmark C Programs

In the third set, we use our LBR decision procedure in F-SOFT [8] to verify industry software programs. We used as benchmarks C programs from public domain and industry, including linux drivers, network application software, and embedded programs in portable devices. We consider 7 of these benchmarks with multiple properties such as *array bound violation*, and *assertions*. These programs[3] have predominately linear operations with sporadic non-linear operations such as /, %, and bit-wise &.

We experimented on same platform as before, with time limit of 1000 sec. We present our results in Table [3], showing comparison among LBR-LE, LBR-EL LBR-LL and

[3] Linear operators and predicates in the corresponding models occur in following ranges: $\#(+) : 22 - 81$; #($*_c$): $380 - 1546$; $\#(\leq) : 30 - 80$; $\#(=) : 346 - 1573$; $\#(ITE) : 531 - 1573$. There is at most one non-linear operators /, % and bitwise &. Also, control states and program variables in these models range from 318 to 1513, and 40 to 102, respectively.

BRD approaches. We did not compare with Z3 as we could not integrate Z3 as back-end solver due to porting issues.

Column 1 reports the examples with the number of properties in braces; Columns 2-4, 5-7, 8-10, and 11-13 report the result details for LBR-LE, LBR-EL, LBR-LL and BRD approaches, respectively. Specifically, Column 2 reports the time to find the last witness in sec (T_{LW}), Column 3 reports the BMC unrolled depth reached just before time out (D_{TO}), and Column 4 reports the number of witnesses (#w) found. Similar description applies for Columns 5-7, 8-10, and 11-13, respectively.

Discussion: Observe that LBR-LL approach finds more witnesses or searches deeper compared to the other approaches. Though LBR-LL approach did not solve all the properties, yet the strategy looks quite promising compared to the rest.

Table 2. Comparison on gcd/lcm problems

Ex gcd-p1-p2, lcm-p3-p4	LBR-LE X : L, Y: E T(sec)	LBR-EL X : E, Y: L T(sec)	LBR-LL X : L, Y: L T(sec)	BRD [6] X : L, Y: L T(sec)	# I	Z3 [31] T(sec)
gcd-1891-61	27.9	11.4	12.1	4.6	7	497
gcd-21-3	1.1	1.4	0.8	2.4	3	7
gcd-273-13	5.1	3.0	2.4	2.9	5	26
gcd-7747-127	54.6	19.6	3.6	9.1	8	77
gcd-91-13	2.6	3.4	2.9	2.8	5	20
lcm-1891-1830	123.9	51.0	137.7	333.8	11	TO
lcm-21-14	15.4	7.7	10.5	9.8	4	TO
lcm-273-260	70.6	35.6	37.5	56.2	9	TO
lcm-91-78	20.4	13.8	21.0	27.1	7	TO

Note:

1. gcd-p1-p2: Find x,y s.t $(x + y = p1)$ and $GCD(x, y) = p2$
2. lcm-p3-p4: Find x,y s.t $(x + y = p3)$ and $LCM(x, y) = p4$

Table 3. SMT-based BMC using LBR on industry programs

Ex (#prp)	LBR-LE X : L, Y: E			LBR-EL X : E, Y: L			LBR-LL X : L, Y: L			BRD [6] X : L, Y: L		
	T_{LW}(s)	D_{TO}	#w	T_{LW}(s)	D_{TO}	#w	T_{LW}(s)	D_{TO}	#w	T_{LW}(s)	D_{TO}	#w
B1(88)	69.0	138	40	69.8	138	40	30.3	241	40	30.3	241	40
B2(12)	143.6	24	2	5.1	303	7	7.2	436	7	6.6	399	7
B3(33)	67.3	12	1	962.1	232	22	821.6	261	23	772.2	234	22
B4(14)	-	8	0	51.8	363	10	26.1	530	10	45.7	386	10
B5(57)	-	29	0	878.9	181	17	946.6	184	18	981.2	173	15
B6(5)	831.1	119	2	31.5	467	3	5.7	590	3	24.5	532	3
B7(1)	184.1	161	1	4.2	161	1	1.2	161	1	1.6	161	1

Note:

L:Lazy, E: Eager, -:Not Applicable, T_{LW}: Time to last witness, D_{TO}: depth reached before timeout, #w: number of witnesses found

9 Conclusion

We presented a novel linearization of non-linear operations on bounded integers using bitwise structural decomposition to generate a SMT($\mathcal{LIA}$) decision problem. Such an encoding is specially geared towards guiding SMT($\mathcal{LIA}$) solver for faster search. For the encoded problem, we also presented a novel lazy bounding and refinement LBR algorithm that generates a partially constrained formula with bound constraints that are tightened or relaxed incrementally, as determined from the satisfiability results from the previous runs. Such a procedure avoids re-encoding of the formula and exploits the capability of the state-of-the-art SMT($\mathcal{LIA}$) solver to assert/retract constraints efficiently. We also presented linearization criteria that help in reducing inherent overhead in linearizing non-linear operations. We found in our experiments that our linearization and LBR algorithm are quite efficient and effective in handling decision problems with non-linear operations, compared to the previous approaches based on linearization or the state-of-the-art SMT($\mathcal{BV}$) solver. We conclude that, from a practical viewpoint, such an approach for handling non-linear arithmetic is crucial for both hardware and software verification methodologies.

Acknowledgement

We thank Weihong Li for carrying out the experimentation with Z3.

References

1. Singerman, E.: Challenges in making decision procedures applicable to industry. In: Proc. of Pragmatics of Decision Procedures in Automated Resonings (2005)
2. Ganai, M.K., Gupta, A.: SAT-based Scalable Formal Verification Solutions. Springer Science and Business Media, Heidelberg (2007)
3. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
4. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propogation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
5. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: Proc. of ICCAD (2006)
6. Bozzano, M., Bruttomesso, R., Cimatti, A., Franzén, A., Hanna, Z., Khasidashvili, Z., Palti, A., Sebastiani, R.: Encoding RTL Constructs for MathSAT: a Preliminary Report. In: Proc. of Logic Programming and Automated Reasoning (2006)
7. de Moura, L., Bjorner, N.: 3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
9. Fallah, F., Devdas, S., Keutzer, K.: Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In: Proc. of DAC (1998)
10. Brinkmann, R., Drecshler, R.: RTL-Datapath Verification using Integer Linear Programming. In: Proc. of ASPDAC (2002)
11. Zeng, Z., Kalla, P., Ciesielski, M.: LPSAT: A Unified Approach to RTL Satisfiability. In: Proc. of DATE (2001)
12. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-Based Satisfiability Solving of Presburger Arithmetic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004)
13. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding Bit-Vector Arithmetic with Abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
14. Andraus, Z.S., Sakallah, K.A.: Automatic abstraction and verification of verilog models. In: Proc. of DAC (2004)
15. Seshia, S., Lahiri, S.K., Bryant, R.E.: A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In: Proc. of DAC (2003)
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proc. of DAC (2001)

17. Ganai, M., Ashar, P., Gupta, A., Zhang, L., Malik, S.: Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In: Proc. of DAC (June 2002)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
19. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 547–560. Springer, Heidelberg (2007)
20. Babic, D., Hutter, F.: Spear Theorem Prover. In: Theory and Applications of Satisfiability Testing (2007)
21. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The Bit-level Analysis Tool. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 303–306. Springer, Heidelberg (2007)
22. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
23. Babic, D., Musuvathi, M.: Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Reserach Redmond (2005)
24. Maxima Development Team. Maxima, a Computer Algebra System, http://maxima.sourceforge.net
25. Dantzig, G.B.: Linear Programming and its Extensions. Princeton University Press, Princeton (1963)
26. Badros, G., Borning, A., Stucky, P.: The Cassowary Linear Arithmetic Constraint solving algorithm. In: ACM Transactions on Computer-Human Interaction (2001)
27. Ganai, M.K.: Conference notes, http://www.nec-labs.com/~malay/notes.htm
28. Zaks, A., Shlyakhter, I., Ivančić, F., Cadambi, S., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Using range analysis for software verification. In: International Workshop on Software Verification and Validation (2006)
29. Cormen, T.H., Leiserson, C.E., Rivest, R.H.: Introduction to Algorithms. MIT Press, Cambridge (1989)
30. SRI. Yices: An SMT solver, http://fm.csl.sri.com/yices
31. Microsoft. Z3: SMT solver, http://research.microsoft.com/projects/Z3/
32. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

Evaluating Workloads Using Comparative Functional Coverage

Yoram Adler[1], Dale Blue[2], Thomas Conti[2], Richard Prewitt[2], and Shmuel Ur[1]

[1] IBM Haifa Research Lab, University Campus, Carmel Mountains, Haifa, 31905, Israel
[2] IBM Systems & Technology Group, 2455 South Rd., Poughkeepsie, NY 12601
adler@il.ibm.com, dblue@us.ibm.com, tconti@us.ibm.com, prewitt@us.ibm.com, ur@il.ibm.com

Abstract. We introduce comparative functional coverage – a technique for comparing the coverage of multiple workloads – and the tool in which it was implemented. The need to compare workloads and the use of functional coverage as a technique to explore data are not new. However, the use of functional coverage for comparing workloads has not been addressed as an answer to this long unanswered need. We describe our work in augmenting a functional coverage tool so it can handle multiple data sources. We present the data and an experiment that shows the usefulness of this method.

1 Introduction

Information Technology (IT) customers continuously place pressure on vendors of software, hardware, and services to provide the highest availability and quality of their respective products and services. The IT customer set is extremely large and diverse. Their environments are complex conglomerations of legacy systems and new, state-of-the-art web-based applications. The growth is fast and furious.

Two traditional techniques used to examine the 'profile' of critical customers are: the review of defect escapes and their root cause, and face-to-face customer review and technical exchange.

Defect escapes (found by the customer) are the most logical area to attack in order to better understand how test coverage can be improved. This is because the results are immediate and once the defect is known, the test case that was missed can immediately be created or an existing test workload can be modified to target that area to ensure the issues do not reoccur. This effort is time consuming and is a function of the total number of escapes and the complexity of each escape. Similar approaches have been taken in related work [1], where the defect analysis is performed after development and during release of the product in question. Factors such as high impact defects or more-frequent low impact defects are considered during defect escape analysis [2].

Face-to-face customer review and technical exchange with the vendor are another proven method for extracting important information about the customer's configuration, environment, operational practices, and business requirements. This exchange implements a thorough analysis of business applications and those applications' explicit exploitation of the software or hardware solutions in question. These reviews provide an excellent venue for dialog between the hardware and software providers and the customer. The information retrieved from these discussions can definitely be used by a motivated test team. However, the challenge lies in determining which configurations, environments, applications, and practices should be deployed into the hardware and software vendor's test topology. Even though there is some duplicity in customer's IT shops, each customer's unique set of business applications creates combinations that the IT vendor cannot afford to supply.

These techniques provide us with a number of pieces to the puzzle. However, they cannot improve quality sufficiently, whether alone or together. There is still something missing from the puzzle of ensuring quality coverage.

To address this problem we needed to improve the 'profile' of customers and make our testing results stronger and more predictable. Over the past decade, several teams have attempted to get a handle on the 'profile' of their customers so they can better align their test environments, configurations, and workloads for customers. Some teams profile the customer's workload by generating a set of queries that help determine the desired features for workload optimization [3].

In System z, the test team used a method known as workload profiling. This method retrieves empirical system data produced by the z/OS operating system and performs a logical and statistical comparison between that data from the customer's environment and associated data from IBM's System z test environments. This comparison shows gaps in stress levels against certain functions within the system and the overall system in general. It also shows configuration options chosen by customers and a certain amount of functional flow of control. This methodology has been, and continues to be, utilized both proactively and reactively with some of IBM's largest enterprise customers. It has provided key findings to test organizations within IBM, which resulted in recommendations for specific improvements to their testing techniques and newly discovered defects.

The empirical system data currently available on System z includes interval-based data and logging records. The interval-based data are primarily focused on a specific component or area of the system. It helps provide information related to stress and load, and configuration. The logging capability allows components to trace their module flow of control and identify exceptional situations that have been encountered. In either case, this data provides only a single dimension of the data (component-oriented) – which gives us very little understanding of this component's interactions with others in the whole composition of the system.

With that in mind, we are faced with the challenge of breaking through this single dimensional outlook. This paper discusses a technique and methodology that provides a better understanding of comparative functional coverage or profiling. With this new technique, test teams will be able to further refine their test workloads, environments, and configurations. They will also be able to explicitly reassure customers that organizations can look at their specific IT environment and change the test approach accordingly to better align with the way they use our products.

2 Workload Profiling with Empirical Data

Our efforts to use empirical data for workload profiling began several years ago. The process has been used at multiple customer engagements where data collection, analysis, and test workload improvements have resulted in the removal of new and unique defects. Both of the empirical data opportunities with System z, interval-based system data and event-driven logging data, have been utilized in these efforts. Of the two, the interval-based data mining and analysis tools are the most mature. Collecting the data over an interval acts as an equalizer of the customer and system test data, allowing the computation of standard statistical functions for activity in each interval. The comparison of these statistics forms the basis of our analysis. The minimum and maximum values observed for data points over the set of intervals represent the high and low water marks for each data point. The mean and standard deviation of the same data points provide a measure of the level of flux for the activity, identifying either steady state or highly variable workload activity. Using event-driven data in a similar manner requires an additional step of first being normalized into time intervals before undergoing similar statistical analysis.

We used the Systems Management Facility (SMF) data records from the z/OS environment as the interval-based data for our analyses. They provide an expansive number of data points covering many system components. Handling of the large amount of data involved in analyzing these records required us to develop tooling in two areas: data reduction and data presentation.

The SMF data exists as binary data records that must be parsed and formatted according to the data type of each data point in the record. Because the structure of SMF records are self describing, we were able to write generic programs to de-construct the SMF records and produce a readable value for each data point at each interval.

Once parsed and formatted only the numeric data may used as a source for the calculation of the basic statistics. No analysis is attempted for non-numeric data. Using the customer data as the base, we performed a comparison of statistics between the customer data and a set of system test data on each data point. This comparison is quantified as the percent change or percent difference using the following formula:

$$\frac{\text{compare data point value}_i - \text{base data point value}_i}{\text{base data point value}_i} * 100 = \text{percent change}_i \qquad (1)$$

We then used the above calculation for a set of data points to build web pages for visual analysis. A single web page allows comparison of a set of customer data points to the corresponding set of system test data points. Colors highlight the magnitude of the difference between the base and compare data for each data point. Shades of green indicate test matching or exceeding customer usage while shades of yellow and red areas indicate that test usage is deficient. To speed analysis, a custom report is also available that displays only deficient data points.

The color scheme was successful in quickly evaluating individual data points, but a comparison of the combined data points was not easy. Understanding individual data points and taking corresponding action to address test deficiencies is key to the process. It is also essential to have a summary of the quality of a particular set of test data compared to the customer data. This creates useful information regarding which of several test environments have the closest match to the customer environment. It shows when workload improvements enable data point activity to converge closer to the activity shown in the customer data. To address this need we developed a single 'overall analysis score' for each comparison. Three factors are used in computing the overall analysis score.

1. Data point value factor – a value from 0 to 3 that weighs the importance of the data for the analysis. A higher value assigns more weight to that data point.
2. Data point score factor – a value from 0 to 3 that is assigned based on the following set of classes.

Table 1. Data point score factors

Score Class	Score Factor
Both are equal	0
One is slightly larger than the other	1
One is much larger than the other	2
One is zero and the other is not	3

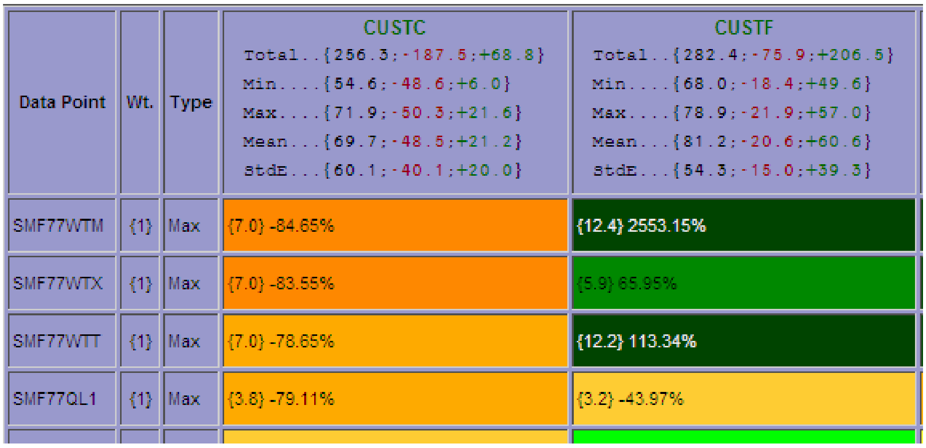
3. Data point magnitude of difference – is used in a log function to weight those data points of greatest difference.

Using the above factors, the data point analysis score is computed as follows:

$$\text{analysis score}_i = \text{value factor}_i \bullet \text{score factor}_i \bullet \log(1 + |\text{difference}_i|) \quad (2)$$

The overall analysis score is the sum of the individual data point scores. The lower the overall analysis score, the more desirable. On the example profiling analysis web page in Figure 1 CUSTF has the best score with a value of 282.4. After changes are made to a test run, the overall analysis score can quickly determine how effective those changes were in more closely matching the customer workload.

It is also possible to extend these same concepts to ctrace records. Typical ctrace data includes information about modules and entry points. Each time a module is



Data Point	Wt.	Type	CUSTC
			Total..{256.3;-187.5;+68.8}
			Min....{54.6;-48.6;+6.0}
			Max....{71.9;-50.3;+21.6}
			Mean...{69.7;-48.5;+21.2}
			StdE...{60.1;-40.1;+20.0}
				CUSTF
				Total..{282.4;-75.9;+206.5}
				Min....{68.0;-18.4;+49.6}
				Max....{78.9;-21.9;+57.0}
				Mean...{81.2;-20.6;+60.6}
				StdE...{54.3;-15.0;+39.3}

SMF77WTM	{1}	Max	{7.0} -84.65%	{12.4} 2553.15%
SMF77WTX	{1}	Max	{7.0} -83.55%	{5.9} 65.95%
SMF77WTT	{1}	Max	{7.0} -78.65%	{12.2} 113.34%
SMF77QL1	{1}	Max	{3.8} -79.11%	{3.2} -43.97%

Fig. 1. Example profiling analysis web page

entered or exited, a ctrace record is written. Instead of being interval-driven, ctrace records are event-driven. Counting the occurrences of each module provides a raw number of times the module was invoked; this can be normalized to invocations per second.

Although our approach does allow the use of ctrace data, it is questionable whether the full potential of the trace data is being exploited. Ctrace data carries with it information beyond the raw counts of module invocations. The order of the entries may reveal common call patterns between modules. Or it may instead be used to look for the opposite, as uncommon call patterns. Depending on other available data in the ctrace record it might be possible to surmise the effect of various input parameters on call patterns. There is also the potential for understanding other calls patterns that may be active in the system and occurring in parallel with the current one. The interval methodology makes it difficult to easily answer new queries that become evident as we analyze a set of data. For this reason we decided to augment our process with a methodology that allows us to look at views of the data not previously considered.

3 Functional Coverage Analysis

Functional coverage [4] [11] is a coverage methodology for evaluating the completeness of testing against application-specific coverage models. Functional coverage is in use in many companies [5], [6] and in many projects. The first and most important step in the functional coverage process is deciding what to cover. The model is usually described in free text, or a story, where some of the words designate attributes. For example the model could be "Lets check that every <unit> sends every possible <signal> to every other <unit>".

The model is usually described as a tuple containing all the attributes described in the story; for example, for our case it will be <unit, signal, unit>. Without the story, the tuple by itself could have many other meanings. For example a different story for the same tuple could be “Lets check that every <unit> received every <signal> while another <unit> has crashed” which may be a reasonable model for checking robustness. Each attribute has a list of possible values. For example, unit may have the values: OS, Mem, Proc. A coverage task is an instantiation of the story with specific values given to the attributes. For every test, we should be able to verify which coverage tasks it covered.

Often some of the tasks in the coverage model are illegal tasks that should not occur. The reasons could be limitations on the inputs (e.g., some units cannot generate some signals) or implementation details (e.g., unit CPU2 cannot send a signal to Proc1). Specifying the illegal tasks is an important part of creating the functional coverage model. It is our experience that bugs are found during this phase since the architect must consider the application from a different view point.

After the coverage model is created, the next step is data collection. In functional coverage, the models are implementation-specific. This causes the coverage tooling, including data collection and data presentation, to be model-specific. Since there are many models, it was not possible to write a good coverage analysis tool for each one. Nevertheless, the coverage processes of different models have much in common—although each model is unique. Tasks have to be updated in tables, regression suites have to be created, and coverage reports on sub-models and on progress have to be made. Once it was realized that implementation-specific models still have many processes that can be automated, a number of functional coverage tools were created to handle all the common requirements. These include tools such as Meteor [7], Comet [4], Specman Elite [8] and Focus [9].

From the outset, functional coverage tool development has concentrated on being able to explore data in variety of ways [10]. This includes the ability to view projections of subsets of the attributes. One such example is the model above, on the sub model <signal, unit> where we check which signals arrived to which units. Another common view looks at a subset of the values, for example, the sub model <signal, unit> when the first unit is not OS. Another very useful view is hole analysis, which automatically discovers large sets of uncovered tasks that have something in common [7].

The ability to explore data easily in a variety of ways is the main reason functional coverage tools are preferred to model-specific tools. It is very easy to conceive of a specific way to view the data and then write a script that will process and present the data in this specific way. However, when you collect the data, you do not know exactly how you will want to look at it. Furthermore, the process of exploring the data is interactive in nature. After half an hour of exploring the data, we usually look at more than ten different views. If we had to write a script for each view, the cost would be prohibitive.

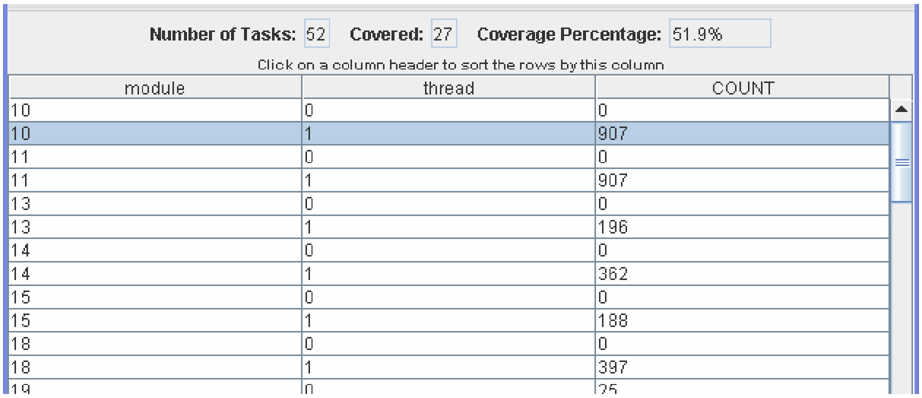
This exploration capability is the reason we decided to use the functional coverage methodology to evaluate the difference between workload used to emulate customer activity and actual workloads collected from the customer. We started the exploration by working concurrently on two equivalent models with two separate input data: one from the customer and one from the client. While it is easy to manipulate each set of data, comparing them is not trivial. For example, it is easy to see how many of each module has been executed by the customer or in test (see Figure 2), but it is not easy to see which of them comprises a larger portion of the relevant workload.

For that reason we enhanced FoCuS, our functional coverage tool with the ability to simultaneously look at multiple data sets. This capability is described in the next section.

4 Comparative Functional Coverage

The idea of comparing two sets of data seems straightforward. Normally there is a coverage model and the traces are compared against the model. The coverage output is composed of a list of tasks, where each task has a description and an indication of how many times it has been covered. Figure 2 shows an example of a coverage report. In this report, each task has two attributes: module and thread. In the table, the second line with the dark background represents a task with a value of 10 for the module attribute and 1 for the thread attribute, which was covered 907 times.

One could add a second data source as shown in Figure 3, giving us the same list of tasks with measurements from two data sources. The measured coverage of Customer data is under the Customer column and the coverage of the Test data is presented under the Test column. At the top, we automatically put tasks with large changes between the sources. The top four rows show tasks covered by only one data source. For example, the first row shows a task that was covered only under the Test data source.

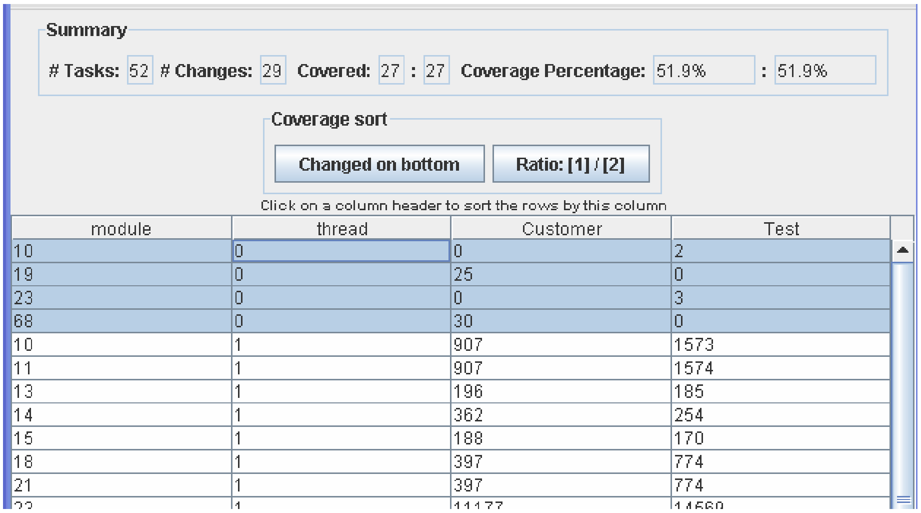


Number of Tasks: 52 **Covered:** 27 **Coverage Percentage:** 51.9%

Click on a column header to sort the rows by this column

module	thread	COUNT
10	0	0
10	1	907
11	0	0
11	1	907
13	0	0
13	1	196
14	0	0
14	1	362
15	0	0
15	1	188
18	0	0
18	1	397

Fig. 2. Coverage results for a single data source



Summary

# Tasks: 52 **# Changes:** 29 **Covered:** 27 **:** 27 **Coverage Percentage:** 51.9% **:** 51.9%

Coverage sort

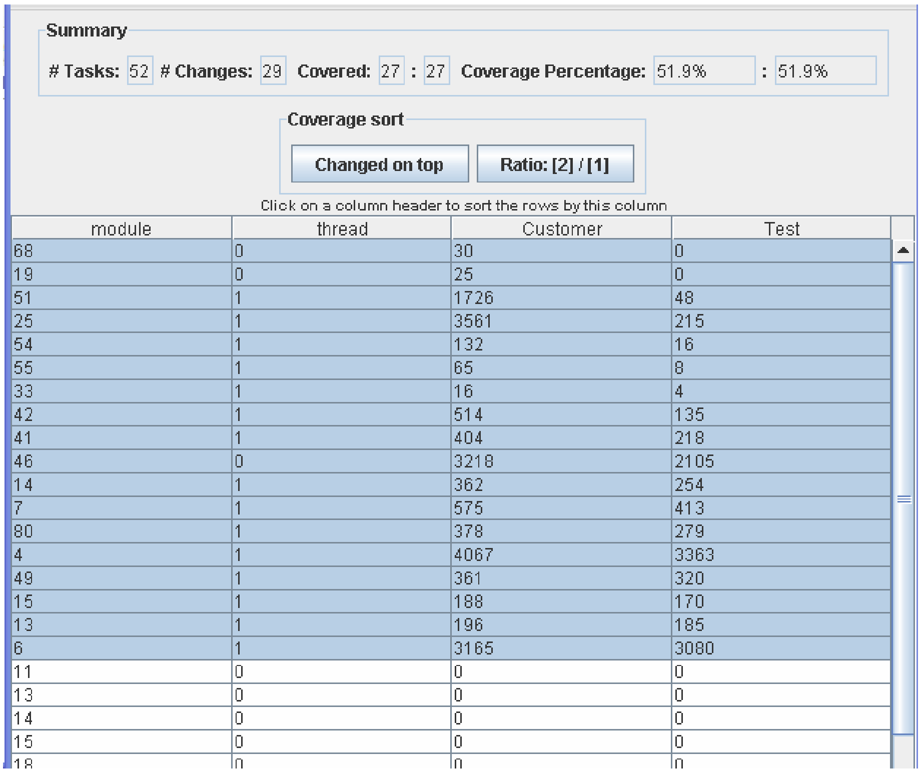
Click on a column header to sort the rows by this column

module	thread	Customer	Test
10	0	0	2
19	0	25	0
23	0	0	3
68	0	30	0
10	1	907	1573
11	1	907	1574
13	1	196	185
14	1	362	254
15	1	188	170
18	1	397	774
21	1	397	774

Fig. 3. Comparing Customer to Test on a functional coverage model

Being able to see results from two data sources requires a number of additional features, such as the ability to sort by relative coverage. Focus tables were able to sort on any one of the columns but this capability is not enough. We initially added two sorts for source1/source2 (**Ratio [1]/[2]** button). Since the sort is reversible, this also adds source2/source1 (**Ratio [2]/[1]** button). For example in Figure 4, the top of the screen shows all the tasks in which the Customer has better coverage than the Test. The bottom of the table (not visible here) shows the complementary tasks which are covered better by the Test than by the Customer.

We added a second sort - **Changed on top** button - that brings to the top the tasks with a high absolute value of change between the sources; this is practical when we



Tasks: 52 # Changes: 29 Covered: 27 : 27 Coverage Percentage: 51.9% : 51.9%

module	thread	Customer	Test
68	0	30	0
19	0	25	0
51	1	1726	48
25	1	3561	215
54	1	132	16
55	1	65	8
33	1	16	4
42	1	514	135
41	1	404	218
46	0	3218	2105
14	1	362	254
7	1	575	413
80	1	378	279
4	1	4067	3363
49	1	361	320
15	1	188	170
13	1	196	185
6	1	3165	3080
11	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
18	0	0	0

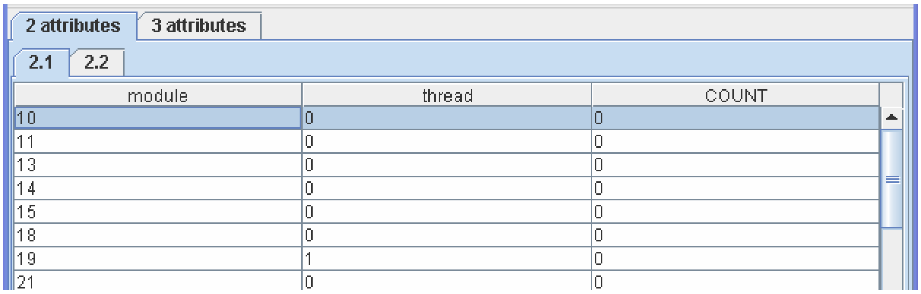
Fig. 4. Tasks sorted by the value of Customer / Test

do not care about the order. This sorting capability enables the user to find out where one workload has concentration which the other lacks. This is very important for the exploration stage and is shown in Figure 3 and is the default of the tool.

Since we are comparing workloads, some functional coverage features were disabled. In our reports, we need to show coverage of the two workloads against a set of tasks. We removed any restriction that changes the task set in an asymmetrical way. If the restriction is oblivious to the tests and depends only on the model, then it can be used. We removed the ability to restrict tasks to those with coverage higher than some number, as the restrictions yield a different set of tasks for each data source and the comparison becomes complicated. For the same reason we do not compare illegal tasks, which are different for each workload.

4.1 Comparative Functional Coverage Holes

The same coverage data can also be shown using another technique called the Subsets (holes) report. Coverage holes are defined as projections on subsets of variables that were not covered. Figure 5 shows an example of the Subsets report. The data

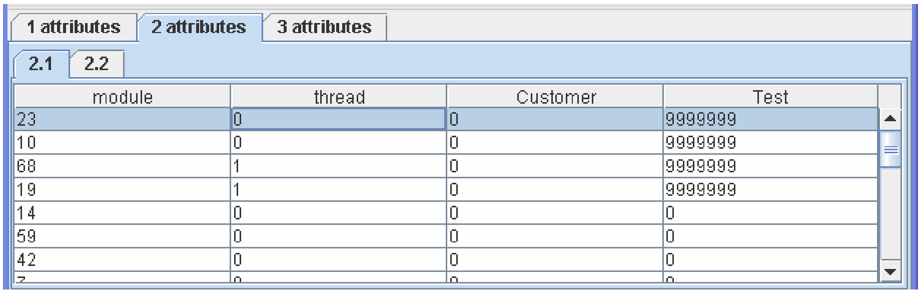


module	thread	COUNT
10	0	0
11	0	0
13	0	0
14	0	0
15	0	0
18	0	0
19	1	0
21	0	0

Fig. 5. Subsets (holes) Coverage result for a single data source

source (Customer) for this report contains tasks that have three attributes ("module", "thread" and "system"). The first row in Figure 5 contains a hole which is a value in this projection ("10", "0"); this hole is never covered, regardless of the value of the third attribute ("system"). Showing holes is a very useful way to present coverage results since it separates the wheat from the chaff.

By adding a second data source, we can have the same description of partial tasks and measure it against two data sources, as can be seen in Figure 6. The measured coverage of Customer data is under the Customer column and one can see the coverage of the Test data under the Test column. The first row in the report shows that the partial task ("23", "0") represents a hole in the Customer data source but not in the Test data source (the "999999" is an indicator for no hole). Similar to the **Comparative Functional Coverage** report, we automatically put at the top the partial tasks with a high absolute value of change between the sources. The top four rows show partial tasks that exist only on the Customer data source.



module	thread	Customer	Test
23	0	0	9999999
10	0	0	9999999
68	1	0	9999999
19	1	0	9999999
14	0	0	0
59	0	0	0
42	0	0	0

Fig. 6. Comparing Customer to Test on a Subsets (holes) coverage result

5 Experimental Results

We validated our comparative functional coverage technique on the SYSSMS component of the z/OS operating system. The data was collected by post processing the logs created by the z/OS ctrace facility; therefore, we did not need to touch the application.

In addition, we had a developer available with deep knowledge to help us interpret the patterns and to understand whether our observations made sense.

We set two goals for the analysis. The first goal was to better understand the functional flow of the SYSSMS modules across the system. The second was to ensure that the test covers the same flows as the customer, at the very least.

The information we gleaned from ctrace for use in our functional coverage model is demonstrated in Figure 7. It included:

1. A numeric module identifier corresponding to the module in the MDID field.
2. A label to identify the entry as either a module ENTRY or a module EXIT.
3. A thread identifier for the request constructed by combining the TCBA and HASID fields.
4. An implied order of the module entries.

As an aside it is interesting to note the thread identifier served as a real life reminder of the iterative nature in building a correct model. Our first pass at the model and analysis brought us to the conclusion that module number 4 was an entry point to the rest of the code. In verifying this with the developer we learned that it was true, but we also learned that the thread identifier for module numbers 46 and 4 could not be trusted. Leaving them in the model caused the thread to sometimes be incorrectly identified, which invalidated many of the other conclusions of the analysis. While modifying the model to restrict the entries for the two modules we also discovered a third module, number 48, for which the thread identifier could not be trusted. The model restrictions had to again be modified to account for this.

```
--------------------------------------------------------------------------
DSYS      ENTRY     00000000  00:19:43.789971  Module Entry
--------------------------------------------------------------------------
ASID: 00E3  COMP: SMSVSAM (DF122)  SCMP: VRM  (097)  FNID: 019  MDID: 021
MDNM: IDAVRSUS  TCBA: 80034000  RETN: B58EDA6A DINM: 0001   HASID:  0159
--------------------------------------------------------------------------
KEY:  6132   LEN:  0014
 +00000000: 00000000 80000000 00000000 00000000  *................ *
 +00000010: 00000000                             *....             *


--------------------------------------------------------------------------
DSYS      EXIT      00000002  00:19:43.792024  Module Exit
--------------------------------------------------------------------------
ASID: 00E3  COMP: SMSVSAM (DF122)  SCMP: VRM  (097)  FNID: 019  MDID: 021
MDNM: IDAVRSUS  TCBA: 80034000  RETN: B58EDD02 DINM: 0002   HASID:  0159
--------------------------------------------------------------------------
KEY:  0098   LEN:  0004
 +00000000: 00000000                             *....             *
KEY:  0006   LEN:  0004
 +00000000: 00000000                             *....             *
```

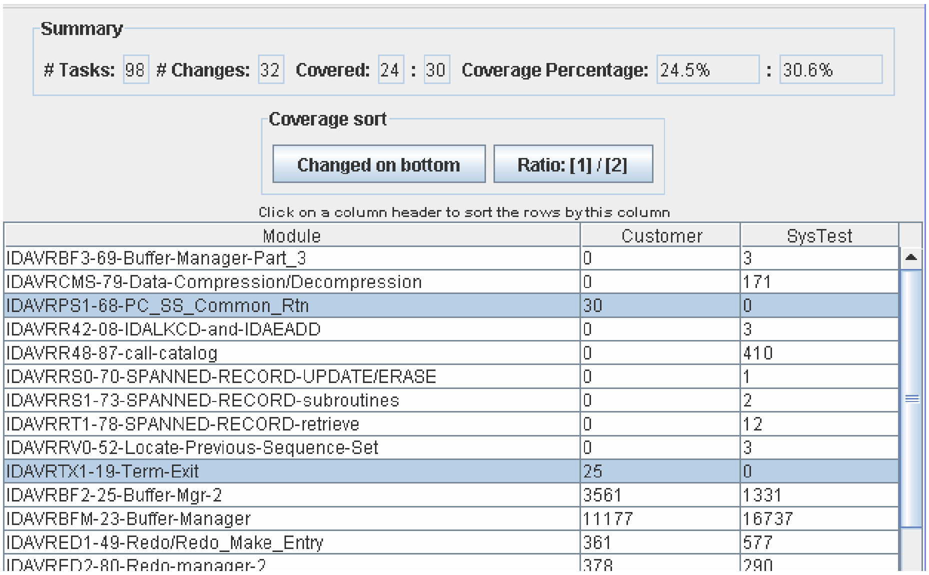
Fig. 7. Example ctrace entries

Our model has six attributes:

1. Module – One of 98 possible module names, translated from the MDID to the associated alphanumeric module name
2. MeOnThread – '1' if the currently starting module is already running on the same thread, otherwise '0'
3. MeOnSystem – '1' if the currently starting module is already running on any other threads, otherwise '0'
4. OtherOnThread – '1' if there are any other modules already running on the same thread, otherwise '0'
5. OtherOnSystem – '1' if there are any other modules already running on any other threads, otherwise '0'
6. MeNextOnThread – '1' if the next starting module on this thread is the same as the currently starting module, otherwise '0'

We performed our analysis using the Focus tool with the two reports described in the previous section: the Coverage report and the Holes Subsets report. The use of restrictions and filters in the reports was important in forming the desired queries.

Our first query was to understand what modules (functions) were being used by the customer vs. test. Of these, the most important were modules that were used by the customer but not used by the test. Our analysis in Figure 8 did in fact show deficiencies in module coverage by test. Modules IDARPS1-68-PC_SS_Common-Rtn and IDDAXRTX1-19-Term-Exit (i.e., modules 68 and 19) were used by the customer but not by the test. It is interesting to note that both the customer and test cover less than one third of the possible modules. This highlights the fact that our goal for using comparative functional coverage for this test phase is to match the customer's usage and not to ensure complete coverage.


Summary

Tasks: 98 # Changes: 32 Covered: 24 : 30 Coverage Percentage: 24.5% : 30.6%

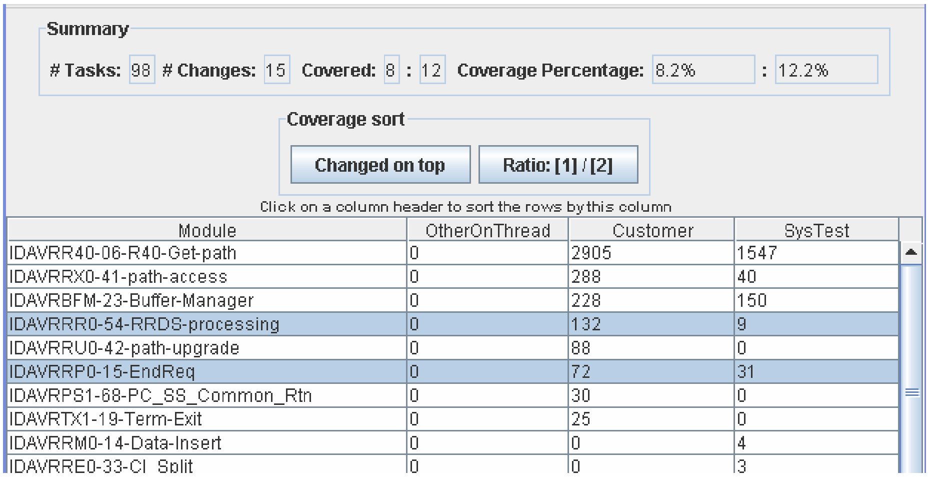
Coverage sort

Click on a column header to sort the rows by this column

Module	Customer	SysTest
IDAVRBF3-69-Buffer-Manager-Part_3	0	3
IDAVRCMS-79-Data-Compression/Decompression	0	171
IDAVRPS1-68-PC_SS_Common_Rtn	30	0
IDAVRR42-08-IDALKCD-and-IDAEADD	0	3
IDAVRR48-87-call-catalog	0	410
IDAVRRS0-70-SPANNED-RECORD-UPDATE/ERASE	0	1
IDAVRRS1-73-SPANNED-RECORD-subroutines	0	2
IDAVRRT1-78-SPANNED-RECORD-retrieve	0	12
IDAVRRV0-52-Locate-Previous-Sequence-Set	0	3
IDAVRTX1-19-Term-Exit	25	0
IDAVRBF2-25-Buffer-Mgr-2	3561	1331
IDAVRBFM-23-Buffer-Manager	11177	16737
IDAVRED1-49-Redo/Redo_Make_Entry	361	577
IDAVRED2-80-Redo-manager-2	378	290

Fig. 8. Comparison of modules used

Our next queries were related to the module interactions. The OtherOnThread and OtherOnSystem attributes allowed us to look for other likely modules that are points for entry to other strings of function. They also served to give us an idea of concurrent processing on the system. In Figure 9 we see that modules 6, 23, and 41 are most likely to start alone on the thread (using the restriction OtherOnThread = 0) for both the customer and the test. However the fourth most likely module to start alone on the thread is different for the two: module 54 for the customer and module 15 for test. This suggests that the customer makes use of some function with more relative frequency than is done by the test. Modules 68, 42, and 19 are also of interest because they are modules that the customer uses, but for which there was no test activity. Modules 68 and 19 had been identified earlier as exposures but 42 had not. Even though module 42 was used by the test, it does not have the same usage pattern as at the customer.



Tasks: 98 # Changes: 15 Covered: 8 : 12 Coverage Percentage: 8.2% : 12.2%

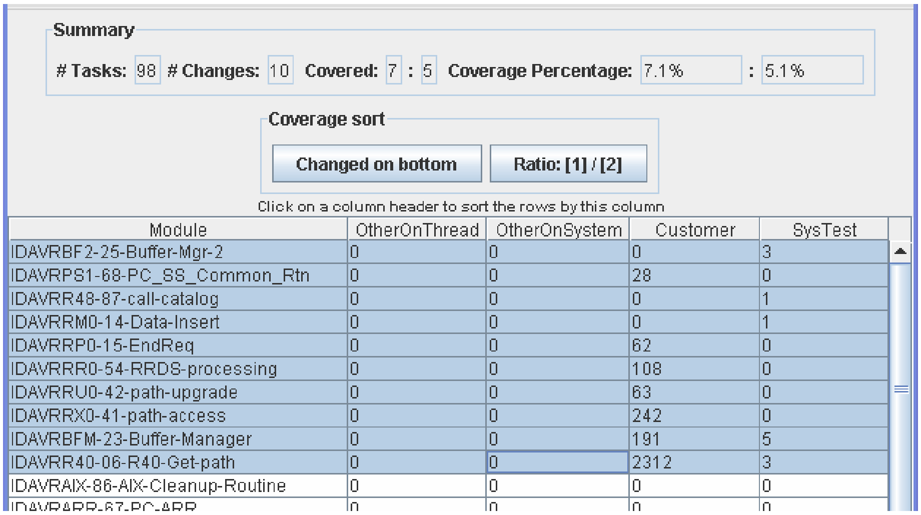
Module	OtherOnThread	Customer	SysTest
IDAVRR40-06-R40-Get-path	0	2905	1547
IDAVRRX0-41-path-access	0	288	40
IDAVRBFM-23-Buffer-Manager	0	228	150
IDAVRRR0-54-RRDS-processing	0	132	9
IDAVRRU0-42-path-upgrade	0	88	0
IDAVRRP0-15-EndReq	0	72	31
IDAVRPS1-68-PC_SS_Common_Rtn	0	30	0
IDAVRTX1-19-Term-Exit	0	25	0
IDAVRRM0-14-Data-Insert	0	0	4
IDAVRRE0-33-CI_Split	0	0	3

Fig. 9. Alone on thread

Expanding the query to include other threads in the system allows us to look for modules that are starting alone in the system and gives us an idea of the amount of concurrent processing that is occurring. In Figure 10 we see that the customer is much more likely than test to have a module running alone on the system. We interpret this to mean that the customer runs with much lower levels of concurrency.

Applying a similar analysis to the MeOnThread and MeOnSystem attributes shows similar usage patterns. (No figure is shown.) Customer and test have a common set of modules that are most likely to run on the thread, and then branch into different usage patterns. Looking across the system shows much higher counts for MeOnSystem than for test, again indicating a higher level of concurrent processing for test.

The MeNextOnThread attribute counts the number of times a module exits, and is then called again with no other modules appearing between; it applies only to the thread level. Our test analysis of this attribute revealed that both the customer and test have modules that exhibit this behavior, with some modules that behave differently between the two. (No figure is shown.)



Summary

Tasks: 98 # Changes: 10 Covered: 7 : 5 Coverage Percentage: 7.1% : 5.1%

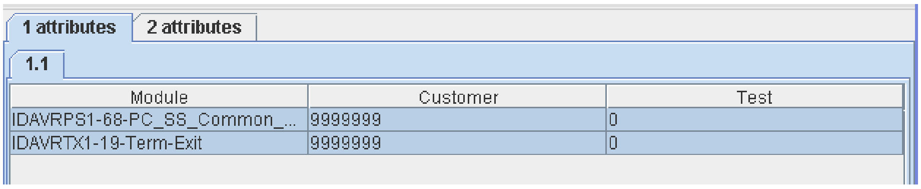
Coverage sort

Click on a column header to sort the rows by this column

Module	OtherOnThread	OtherOnSystem	Customer	SysTest
IDAVRBF2-25-Buffer-Mgr-2	0	0	0	3
IDAVRPS1-68-PC_SS_Common_Rtn	0	0	28	0
IDAVRR48-87-call-catalog	0	0	0	1
IDAVRRM0-14-Data-Insert	0	0	0	1
IDAVRRP0-15-EndReq	0	0	62	0
IDAVRRR0-54-RRDS-processing	0	0	108	0
IDAVRRU0-42-path-upgrade	0	0	63	0
IDAVRRX0-41-path-access	0	0	242	0
IDAVRBFM-23-Buffer-Manager	0	0	191	5
IDAVRR40-06-R40-Get-path	0	0	2312	3
IDAVRAIX-86-AIX-Cleanup-Routine	0	0	0	0
IDAVRARR-67-PC-ARR	0	0	0	0

Fig. 10. Alone on system

As referenced earlier, Focus offers specific automation to aid in holes analysis. Holes analysis in functional coverage normally refers to any function that is not covered. In our comparison of customer and test, holes takes on a narrower meaning and refers to behaviors exhibited by customer but not test. For example, the Holes Subsets report in Figure 11 identifies the modules which are used by the customer but not by test. Using the holes report we can quickly see the same modules 68 and 19 that we identified earlier with the Coverage report.

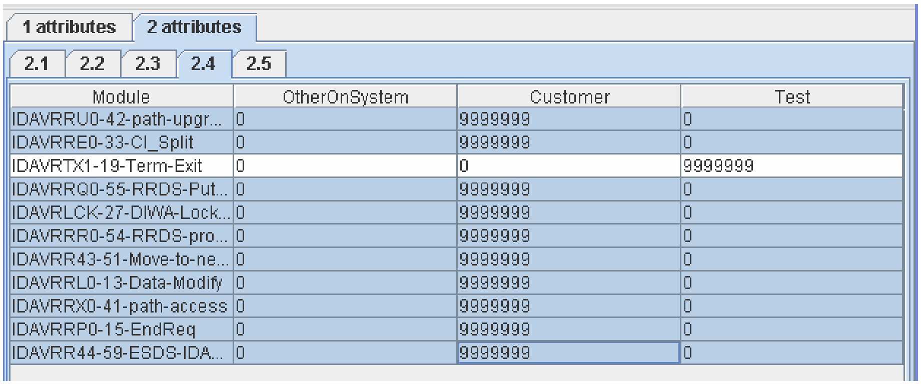


Module	Customer	Test
IDAVRPS1-68-PC_SS_Common_...	9999999	0
IDAVRTX1-19-Term-Exit	9999999	0

Fig. 11. Module holes

As an example of looking for holes across more than one attribute at a time, the combination of Module and OtherOnSystem has not been discussed above but reveals multiple holes in the test coverage as compared to the customer. This is shown in Figure 12.

In the above examples we used the model data in two different ways. First we inferred operational characteristics of the customer and test workloads. Our conclusions about levels of concurrency are an example of this. Second, we identified specific characteristics of the customer which are not exhibited by the test. Modules used



Module	OtherOnSystem	Customer	Test
IDAVRRU0-42-path-upgr...	0	9999999	0
IDAVRRE0-33-CI_Split	0	9999999	0
IDAVRTX1-19-Term-Exit	0	0	9999999
IDAVRRQ0-55-RRDS-Put...	0	9999999	0
IDAVRLCK-27-DIWA-Lock...	0	9999999	0
IDAVRRR0-54-RRDS-pro...	0	9999999	0
IDAVRR43-51-Move-to-ne...	0	9999999	0
IDAVRRL0-13-Data-Modify	0	9999999	0
IDAVRRX0-41-path-access	0	9999999	0
IDAVRRP0-15-EndReq	0	9999999	0
IDAVRR44-59-ESDS-IDA...	0	9999999	0

Fig. 12. Holes - other on system

by the customer and not by the test, along with other holes analysis, fit into this category. In summary, using our model along with the FoCuS tool was successful in identifying instances of both categories.

6 Conclusions

Creating a representative workload is a very difficult problem and there are many potential dimensions along which one workload can try to emulate another. It is very easy to miss the forest for the trees.

Previously, we tackled the problem by choosing dimensions or evaluation criteria, measuring them both at the test and the customer, and comparing the results. We had literally thousands of dimensions. We would score each, bias according to importance, and then summarize the costs to give an evaluation to the workload. This approach proved very useful but was lacking in flexibility. We measured what we set out to measure but we had no way to explore the data looking for unexpected trends and behaviors in which the customer and test behave differently.

In order to reduce the amount of work done to collect the data (instrumentation and special scripting) and to enable exploration of the data, we decided to augment the functional coverage methodology for comparative functional coverage.

In this work, we described the features of a comparative functional coverage tool and the kind of analysis it can carry out. We then showed real data and the capabilities of a general functional coverage data exploration tool in finding trends and behaviors that distinguish the test from the customer.

We found this work very useful and we are already seeing new uses for these capabilities with many of our customer engagements. We believe that this methodology will not only make it into functional coverage tools but will also be incorporated within code coverage and additional testing tools.

References

1. Leszak, M., Perry, D.E., Stoll, D.: A Case Study in Root Cause Defect Analysis. In: 22nd International Conference on Software Engineering, p. 428 (2000)
2. Mockus, A., Zhang, P., Li, P.: Drivers for Customer Perceived Software Quality. In: 22nd International Conference on Software Engineering, pp. 225–233. ACM Press, St. Louis (2005)
3. Golfarelli, M., Saltarelli, E.: The Workload You Have, the Workload You Would Like. In: 6th ACM International Workshop on Data Warehousing and OLAP, pp. 79–85. ACM Press, New York (2003), http://doi.acm.org/10.1145/956060.956075
4. Grinwald, R., Harel, E., Orgad, M., Ur, S., Ziv, A.: User Defined Coverage—a Tool-Supported Methodology for Design Verification. In: Proceedings of the 35th Annual Conference on Design Automation (1998)
5. Coverage-Driven Functional Verification: Using Coverage to Speed Verification and Ensure Completeness. Verisity Design, Inc. (2001), Retrieved from: http://www.verisity.com/resources/whitepaper/coverage_driven.html
6. Gluska, A.: Coverage-Oriented Verification of Banias. In: Proceedings of the 40th Conference on Design Automation, pp. 280–285. ACM Press, New York (2003), http://doi.acm.org/10.1145/775832.775906
7. Lachish, O., Marcus, E., Ur, S., Ziv, A.: Hole Analysis for Functional Coverage Data. In: Proceedings of the 39th Conference on Design Automation (2002)
8. Specman tool, Retrieved from: http://www.verisity.com/products/specman.html
9. FoCuS tool, Retrieved from: http://www.alphaworks.ibm.com/tech/focus
10. Azatchi, H., Fournier, L., Marcus, E., Ur, S., Zohar, K.: Advanced Analysis Techniques for Cross-Product Coverage. IEEE Trans. Comput. 55(11), 1367–1379 (2006)
11. Piziali, A.: Functional Verification Coverage Measurement and Analysis. Springer, Heidelberg (2004)

Iterative Delta Debugging

Cyrille Artho

Research Center for Information Security (RCIS), AIST, Tokyo, Japan

Abstract. Automated debugging attempts to locate the reason for a failure. Delta debugging minimizes the difference between two inputs, where one input is processed correctly while the other input causes a failure, using a series of test runs to determine the outcome of applied changes. Delta debugging is applicable to inputs or to the program itself, as long as a correct version of the program exists. However, complex errors are often masked by other program defects, making it impossible to obtain a correct version of the program through delta debugging in such cases. Iterative delta debugging extends delta debugging and removes series of defects step by step, until the final unresolved defect alone is isolated. The method is automated and managed to localize a bug in some real-life examples.

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [8,10]. Automated test runs allow efficient verification of software as it evolves. However, in the case of failure, defect localization (debugging) is still a largely manual task.

In the last few years, several automated debugging techniques have been developed in order to facilitate fault-finding in complex software [5,13]. Delta debugging (DD) is one such technique [7,14]. It takes two sets of inputs, one which yields a correct result and another one which causes a failure. DD minimizes their difference while preserving the successful test outcome. DD can be applied to the program input or the source code of the program itself, using two different revisions of the same program. The latter variant treats the source code as an input to DD. It therefore includes a compile-time step, which produces mutations of the program to be analyzed, and a run-time step, where the outcome of that mutation is verified by testing.

DD obtains an explanation for a test failure. The scenario considered here is where a test fails for the current version. If an older correct version exists, DD can be used to distill the essential change that makes the new version fail on a given test, and thus reduce a large change set to a minimal one [14]. DD is applicable when there exists a known version that passes the test. For newly discovered defects, this may not be the case. For such cases, we propose iterative delta debugging (IDD). The idea is based on the premise that there exists an old version that passes the test in question, but older versions of the program may have other defects introduced earlier that prevent them from doing so. By successively back-porting fixes to these earlier defects, one may eventually obtain a version that is capable of executing the test in question correctly [1].

IDD yields a set of changes, going back to previous revisions. The final change, applied to the oldest version after removal of earlier defects, constitutes an explanation for the newest defect. Furthermore, the same algorithm that is used to back-port fixes to older versions can also serve to port the found bug fix forward to the latest revision (or a version of choice). Our approach is fully automated. We have applied the algorithm to several large, complex real-world examples in different programming languages and types of repositories. Even though it was not known a priori whether a working revision could be found, we have found working revisions in some cases.

This paper is organized as follows: Section 2 introduces the idea behind IDD. DD and IDD are described in Sections 3 and 4, respectively. The implementation and experiments carried out are described in Sections 5 and 6. Section 7 concludes, and Section 8 outlines future work.

2 Intuition Behind IDD

Developers have used change sets for debugging before. IDD automates this process:

1. A test fails on the current version. It is assumed to have passed on an older version.
2. By successively going back to older versions, one tries to obtain a version that passes the test.
3. One tries to distill a minimal difference between the "good" and the "bad" version, which constitutes the change that introduced the defect.

Step 2 tries to isolate two successive revisions, one that passes a test, and another one that fails. Step 3 attempts to minimize the difference between these two revisions. In that step, we assume that delta debugging (DD) is used to minimize a given change set while preserving the given test outcome [14].

If a version passing the test (a "good" version) is known a priori, then the search for the latest defective version can be optimized by using binary search instead of linear search. This idea has been implemented in the source code management tool `git`, which is used to maintain the Linux kernel sources [11].

The process becomes more complex when the "good" version is not known. Assume there exists a test that fails on a current version. We will call this outcome *fail*. A correct result is denoted by *pass*. Besides these two outcomes, there may also be different incorrect outcomes of the test, denoted by *err*. A set of changes between two versions of a program is referred to as a *patch*.

A test case may not be applicable to older revisions due to missing features or other defects that prevent the test from executing properly. In this case, IDD utilizes DD to apply the necessary changes from a newer version to an older version, to allow a test to run. Figure 1 shows how IDD builds on DD. IDD starts from a version that fails (version 4 in Figure 1). Unlike in the original scenario for DD, a version that passes is not known a priori. IDD successively goes back to previous versions and assesses whether the same failure persists. If the test outcome differs, DD is applied to the last failing version and the older version. This identifies the source code change that made the old version behave differently. One IDD iteration thus either (a) skips a version where the outcome does not change, (b) finds a correct version, or (c) eliminates a older defect (*err*) that covered a newer one.

In the example in Figure 1, version 3 produces case (c): It does not pass the test, and even fails to reproduce the behavior of version 4. DD is then applied between versions 3

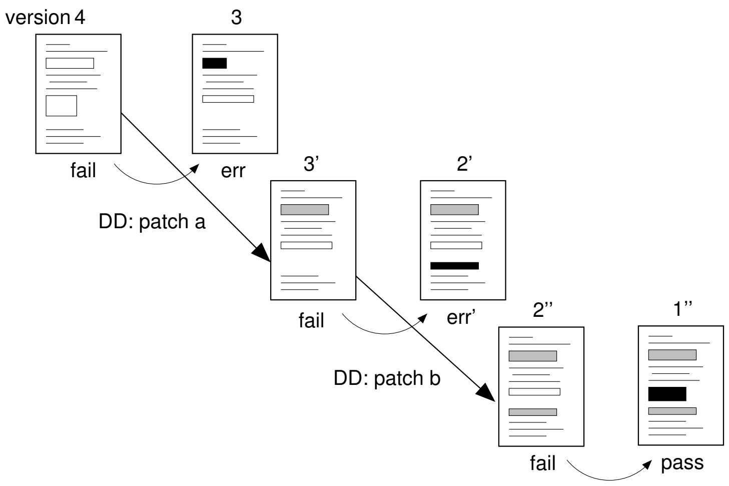


Fig. 1. Iterative delta debugging

```
prev_version = current_version = latest_version();
patch = {}
original_result = current_result = test(current_version);
while (current_result $\neq$ $pass$) {
   current_version = predecessor(current_version);
   current_result = test(current_version $\oplus$ patch);
   if (current_result $\neq$ original_result) {
      patch = DD(current_version, original_version);
   }
   prev_version = current_version;
}
return patch;
```

Fig. 2. IDD algorithm in pseudo code

and 4. The resulting minimal change, patch a, can then be applied as a patch to version 3, fixing the earlier defect in version 3, and producing a new version $3'$. The resulting version fails again in the same way as version 4 did. In Figure 1, the change set (patch) that is back-ported is shown by a thick arrow pointing to the correct, changed code.

The iterative process of IDD continues by applying this patch to older version (such as version 2) before running tests. This produces version $2'$, on which the test is run. That version behaves differently from $3'$, so DD is applied again to find the minimal change required to fix the program. This produces a new patch b (the change between $3'$ and 2). This patch usually contains changes of the previous patch a. The resulting new version is therefore called $2''$. After version 2 is repaired, IDD continues. The patched version 1, $1''$, passes, and IDD terminates successfully in this example.

IDD will eventually find a version that passes the test, or run out of older versions (see Figure 2). IDD is fully automatic as long as the test process does not require human intervention. The process starts from a version where the test in question successfully

Step	Change set	Test verdict
1	`00000000`	`fail`
2	`11110000`	`fail`
3	`11111100`	`pass`
4	`11111000`	`fail`
5	`11110100`	`pass`
6	`00000100`	`pass`

Fig. 3. Delta debugging illustrated

compiles, and produces a known, well-defined output. (Even though that output is erroneous, it is important to use it as a measure against changes.) IDD successively tries older revisions, and uses DD as a subroutine upon failure. DD produces a minimal change set from a newer (working but not fully correct) revision will be back-ported to the older one, which produces an error.

3 Delta Debugging

3.1 Delta Debugging

Delta debugging (DD) uses bisection to minimize the difference between two versions of program input while preserving the behavior that the first input generates. It can also be applied to the program source code itself. When applied to program source code, DD typically operates on a description of the difference between two revisions. The difference set generated by the Unix tool `diff` provides this change set readily. It also has the advantage that variations of the change set can automatically be applied by `patch`, followed by compilation and execution of the program.

Delta debugging applies bisection to an entire change set. This evaluates changes in contiguous subsets only, and ignores vast parts of possible subsets of a change set. Since the size of a power set is exponential in the size of the original set, exhaustive evaluation of all possible change sets is impossible. In the following discussion, a “successful” or passing test is a test whose outcome corresponds to the desired value, such as the same kind of test failure observed in a previous revision.

Figure 3 illustrates DD on a simple example. The change set includes eight elements, the presence or absence of each is illustrates by a “1” or “0”, respectively. Let the first element by the leftmost bit in the change set, and the eighth element be the rightmost bit. Initially, the empty change set is tried. The test, when executed unchanged, fails. DD then attempts to isolate the reason of the failure. The assumption is that if the test passes with only the first half of the change set being active, then the reason for the failure must lie in the first half of the change set. Conversely, if the test fails with the second half of the change set disabled, then at least a part of the second half of the change set is necessary for the test to succeed. This is what happened in the example in Figure 3: The test still fails after the initial bisection of the state space. Therefore, at least one element of the last four elements in the change sets is necessary to achieve the desired outcome.

In the next DD step (3), the subset of the last four elements is again bisected, and the last two elements are ignored. As the test passes, they can be safely dropped from the change set. Now, the algorithm backtracks and analyzes the subset containing elements 5 and 6. A direct implementation of backtracking would again analyze the change set of step 2. As that one is known to fail, it can be skipped. The change set in step 4 therefore contains elements 1–5; the test fails, confirming that element 6 is necessary. Step 5 confirms that element 5 can be dropped; further backtracking analyzes the first half of the change set, which again does not contribute to the test outcome in this example.

DD assumes that each element of a change set is independent of all others. As the format of the Unix `diff` tool is line-based, DD works best on data where one line is independent of any other line, such as a flat configuration file. Obviously, program source code does not conform to this property. For local code changes, though, DD provides a good approximation of the minimal change set. DD fails in cases where the hierarchical structure of program source code conflicts with line-by-line analysis of changes. In typical programming languages, a class scope surrounds declarations of functions and methods, which contain declarations of local variables and code. Higher-level statements cannot be removed without removing everything within its scope. Therefore, when trying to eliminate the addition of a new method, DD tends to eliminate statements and declarations but not necessarily the entire method scope. Conversely, when eliminating the removal of particular statements, it is not possible to remove an entire method without removing all the program statements it contains.

Figure 4 shows some of the problems arising with DD. It shows two examples in "unified diff" format, as produced by `diff -u`, with file names and line numbers omitted. On the left hand side, a C function is added to the code. DD will analyze the given changes in conjunction with many other changes, spanning hundreds of lines. Bisection of the state space may happen at any location, and may not coincide with function or method boundaries.

Incomplete functions will not compile, resulting in invalid code subsets whenever the syntactic structure of the target language is violated. Because of this, DD can only manage to remove unused functions if bisection hits the boundaries of its declaration. In Figure 4, for DD to succeed fully, bisection needs to hit the function boundaries and the scope of the preprocessor `#if`/`#endif` construct. In other cases, DD only manages to remove the code inside the function, and, if it works from bottom to top, the declaration of local variable `x`. On the right hand side, where a function is removed, the situation is even worse. A reduction of the change set would correspond to adding code.

Function addition	Function removal
`+ #if 0`	`- static void`
`+ static int`	`- bar(int y) {`
`+ foo (void * data) {`	`-   int x;`
`+   int x;`	`-   x = y;`
`+   x = 42;`	`-   if (y)`
`+ }`	`-     bar(--x);`
`+ #endif`	`- }`

Fig. 4. Different use cases for delta debugging

As statements cannot be compiled without an enclosing function declaration, bisection needs to hit the function boundary, or none of the changes regarding statements can be removed. This example shows why DD is of limited usefulness if large parts of the code, especially function declarations and interfaces, change.

3.2 Hierarchical Delta Debugging

The problem of having to conform to a syntactic structure also occurs when applying DD to hierarchical data, such as XML. Previous work has implemented hierarchical delta debugging to produce correctly nested changes of XML documents [7]. In program code, it is very difficult to obtaining change sets that include their hierarchical scope. Luckily, the changes produced by the `diff` utility contain some hierarchy in it: Change sets consist of changes to individual files, which are in turn broken up into so-called "hunks", which contain a number of line-based changes. Figure 5 illustrates this hierarchy using the "unified diff" format. In real code repositories, some changes are entirely local in the scope of a single file or a block of code where it occurs. Extending DD to include the hierarchy of the generated patches can therefore improve precision of DD in these cases, as shown in the experiments in Section 6.

This hierarchy provides possible boundaries for the state space search. Instead of bisecting the state space in the middle, based on the number of lines involved, bisection proceeds hierarchically, across files, hunks, and lines. First, sets of changes across files as a whole are analyzed. If a change for an entire file cannot be ignored, a more fine-grained search proceeds on hunk level, then on line level.

Usage of the patch file hierarchy improves recognition of local code changes, but it does not take care of interdependent changes. Addition or removal of a function may be taken care of by multiple iterations of DD: Once all calls and references to a function are removed, then the function itself may be removed as well. Unfortunately, even hierarchical DD cannot deal with certain changes affecting multiple files. For example, in cases where the signature of a function changes, its definition and all instances of its usage have to be changed simultaneously. A bisection-based algorithm such as DD cannot isolate such changes and produces overly large change sets.

Patch	Hierarchy
`Index: file1`	File
`@@ -42,2 +42,3 @@`	Hunk
`  context`	
`+ addition`	Line
`  context`	
`@@ -84,4 +85,3 @@`	Hunk
`  context`	
`- removal`	Line
`+ addition`	Line
`- removal`	Line
`  context`	

Fig. 5. The hierarchy of a change set produced by `diff -u`

4 Iterative Delta Debugging

IDD starts with a test failure in a new revision. The goal is to find an older revision that passes the given test. Whenever the outcome of a test changes in a different way than succeeding, e. g., by executing an infinite loop, delta debugging is used to create a minimal patch that preserves the former outcome.[1] Iteration proceeds until no older revisions are available, a revision where the test passes is found, or a timeout is reached.

4.1 Iteration Strategy

A given test case (or a set of test cases) is compiled, executed, and evaluated by a shell script. This evaluation is successively applied to older revisions, until a working version of the code is found. In most cases, the outcome of a particular test does not change from one revision to the next. Instead, older revisions contain changes affecting other parts of the program. Delta debugging is only necessary when the test case of interest changes.

Previous work implemented this iteration strategy but involved manual patch creation [1]. Usage of DD automates our method. Unlike in previous work, patches are not accumulated, but replaced with a new patch each time delta debugging (DD) is invoked.

4.2 Forward Porting

So far, the given algorithm attempts to locate a correct version of the program by going back to older revisions. Patches (minimized by DD) are applied whenever a test cannot be executed in an older version. If the algorithm is successful, it will eventually find a revision where the given test passes. The final change set will contain a bug fix for the given test, as well as all the "infrastructure" needed to execute it. If the bug fix applies to a version that is much older than when the test was implemented, the portion of the change set containing new features that are necessary for a given test may be large. Intermediate design changes may have taken place, making it impossible to directly apply the resulting patch to the current version. However, the primary concern is usually to fix the latest revision of the software rather than an old one. Therefore, after having found a successful revision, IDD is again applied in reverse, going from the correct version to the current one. DD is again invoked as a subroutine whenever a patch cannot be applied. In this way, forward IDD generates a patch for the current version.

5 Implementation

5.1 Compilation and Test Setup

Within each iteration of IDD, and for evaluating changes of a test, IDD uses a set of scripts to automate program compilation and testing. In detail, this involves the following steps:

[1] Even though nested dependencies require multiple DD iterations to be removed, only one iteration of DD is run at each time except at the final step, when a fix-point iteration is performed.

1. Update of the source code to a new revision. This step may fail due to unavailability of the source repository (e. g. due to a network outage) or because an older version is unavailable.
2. Patching the source code. Patching is a likely point of failure, as any major changes in the source code, including formatting changes, make it impossible to apply a patch to a version other than the one the patch was generated for.
3. Configuration, preceded by deletion of all compiled files. Configuration may require re-generating a build file. Deletion of all compiled files is necessary, as the patching process may create files that do not compile. This would result in the object file of the older version (which successfully compiled) being used in a current test run, and falsify test results.
4. Compilation. Also referred to as the build process, this step generates the executable program and may fail because the given version cannot be compiled before or after application of a patch. If a given version does not compile even when unpatched, then it can be skipped.[2]
5. Test execution. Upon successful compilation, given test cases are executed. Test execution failure may be detected by a given test harness, e. g. if a known correct output is not generated by the test case. However, it also frequently happens that tests fail "outside" the given test harness. There are two ways for this to happen:
   (a) Catastrophic failure: Programming errors, such as incorrect memory accesses, may lead to failures that cannot be caught by the test harness. Therefore, the test harness of the application has to be complemented by an "outer" test harness that is capable of detecting such failures.
   (b) Incomplete test output verification: A given revision may not contain the full code to verify the test output. In that case, such a revision may erroneously report a failed test as successful. The outer test harness has to double-check that the lack of a reported failure actually implies success.[3]

Any failure in the steps above is treated as a critical failure that requires the current version to be fixed. The only exception is if a test fails in exactly the same way as a previous revision, in which case it is assumed that the test outcome has not changed. Otherwise, any change in the result of test execution requires invocation of delta debugging.

5.2 Test Evaluation Accuracy

It should be noted that IDD cannot guarantee that a correct revision (which passes a given test case) is detected as such. If DD removes code that does not contribute to a test failure, but is vital for a test to pass, then IDD cannot recognize a future successful test run as such anymore, as the functionality for the test to pass has been removed by DD. Figure 6 illustrates the problem on an example that could be run in C or in Java.

[2] It sometimes occurs that a repository contains a version that cannot be compiled, for instance, because a developer committed only a subset of all necessary changes.

[3] This may be difficult to achieve in practice, because a given output may not contain enough information to judge whether a modified version still achieves the intended result when a test case passes. In our case, we have also monitored the contents of log files and the amount of memory consumed by tests in order to determine whether source code changes produced behavioral changes in a test inadvertedly.

```
void test1() {
  int result;
  result = 0;
  result = runTest();
  assert(result != 0);
}
```

Fig. 6. Example showing a potential code removal that would prevent a test from passing

Let us assume that the test fails in the current revision, i. e., `runTest` returns 0. This fact is preserved when line 4 that calls `runTest` is removed. Unfortunately, this removes the entire test functionality from the program. A different revision where `runTest` returns 1 would not be recognized as being correct. Even worse, if the code is compiled as a C program, the initialization of `result` and the `return` statement may be removed as well. This causes the return value of the test function to be undefined. During execution, the value probably corresponds to the contents of an element of a previous stack frame that occupied the same memory, but such behavior is platform-dependent. The test harness has to be augmented in order to prevent such deviations.

In order to maintain the exact behavior of failing tests, static or dynamic slicing [3,13] could be used. Slicing would ensure that DD does not remove any lines that contribute to the value of the test result. However, slicing tools are not portable across programming languages, and do not scale well to large programs. We have therefore chosen a less stringent approach. In addition to monitoring the output as closely as possible (using an external test harness), memory consumption and time usage are also checked for deviations from the expected previous value. Furthermore, usage of uninitialized data in C programs is prevented by inspecting compiler warnings and the final change set generated by DD. This process is currently not fully automated, but could be automated by using memory checking tools such as valgrind [9].

5.3 Implementation Architecture

The iterative step of IDD, which applies a given patch across several revisions, is implemented as a shell script using the compilation and test setup described above as a subroutine. The script iterates through existing revisions until a change in behavior is detected. After that, it stops, and control is transferred to the DD program.

When used on a subversion code repository, IDD can take advantage of the fact that all versions are globally and consecutively numbered. Stepping through older revisions is therefore trivial. When using CVS, though, global revision numbers are not available. They were recovered by pre-processing the code repository. During this step, a revision counter was assigned to each revision that was more than five minutes apart from the next one. Revisions being less than five minutes apart were regarded as a single version that was committed using multiple CVS invocations.

DD is implemented as a Java program that takes a patch set derived by the Unix `diff` tool as input. It parses the patch file and produces an internal representation of the state space of all possible patch sets. It then iterates over the state space of all possible change sets. Each change set is produced as a modified patch, which is applied

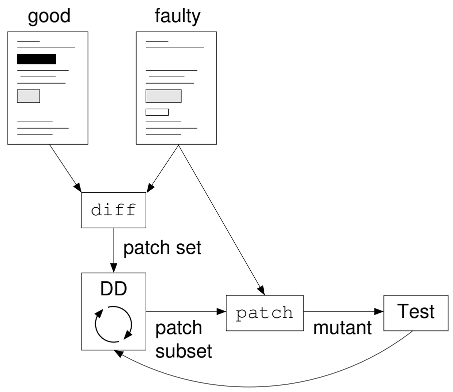


Fig. 7. Implementation architecture of DD

to the faulty version using the `patch` tool. The resulting version reflects a subset of all changes between the good and the faulty version. Compilation and testing are then used to decide whether the mutated version still produces the desired outcome. DD continues its state space search based on this outcome, until the state space is exhausted.

Usage of the Unix `diff` and `patch` tools eliminated the need for a custom representation of the difference between two program versions. This made the DD algorithm rather simple, except for the cache structure, which was difficult to implement for hierarchical DD. While the recursion scheme is elegant to implement, caching adds a subtle interaction between recursion hierarchies of completed, partially completed, and incomplete parts of the search space. To ensure that the state space is bisected correctly even in the case of a nested hierarchy, about 400 unit tests were used. These tests covered many corner cases, but two additional incorrect corner cases were found by inspecting the output when the tool was run.

6 Experiments

Table 1 lists the three example applications taken for the experiments. The projects were chosen according to the following criteria:

- The project had to be sufficiently large to warrant automatic debugging, and have a certain maturity to contain enough older versions to be available for testing.
- The full history of all revisions had to be available.
- Tests had to be fully automated and repeatable, including any configuration files needed to run the test.
- The bugs in question had to be sufficiently well documented to be verifiable, and interesting enough to warrant attention. (This typically meant that the bugs in question had not been resolved for weeks or months.)

For each experiment, the outcome using IDD with "classical" delta debugging and hierarchical delta debugging are shown. Hierarchical delta debugging always fared better than its non-hierarchical counterpart, although the differences vary.

Table 1. Overview of projects used for experiments

Project name	Description	Implementation language	Repository	
			Size [KLOC]	System
Uncrustify	Source code formatter	C++	20	Subversion
Java PathFinder	Java model checker	Java	70	Subversion
JNuke	Java VM/run-time analyzer	C	130	CVS

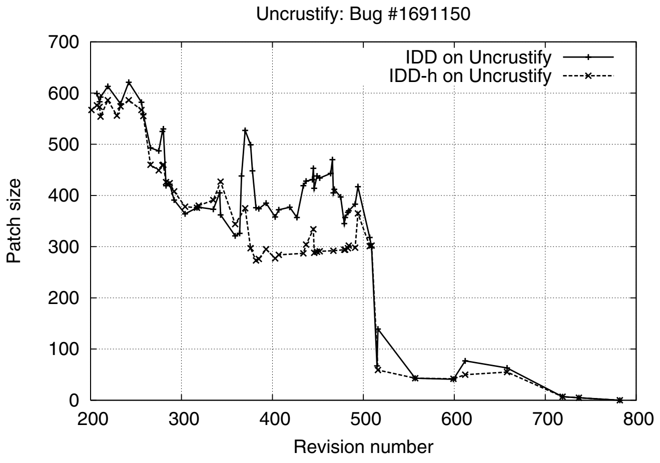


Fig. 8. Result of using IDD on Uncrustify

6.1 Uncrustify

Uncrustify is a source code formatting tool for various programming languages including C, C++, and Java. In addition to formatting code, Uncrustify is also capable of changing the code itself. For instance, one-line statements following an `if` statement do not have to be surrounded by curly braces. Uncrustify can be used to add these optional braces, facilitating future changes. As a case study, bug number 1691150 in the sourceforge bug tracker was used. This bug describes how C++–style one-line comments (starting with `//`) are sometimes incorrectly moved to the wrong code block when they are converted to C-style comments (enclosed by `/*` and `*/`) while optional curly braces are added for `if` and `while` statements.

Figure 8 shows the result of running IDD on that case. The sudden growth of the change set at revision 494 corresponds to a refactoring where the large number of command line options was specified differently in source code. DD could not eliminate this refactoring. As older versions supported fewer and fewer options, the expected growth in the patch set was sometimes compensated by the shrinkage of the part of the patch that concerned command-line options.

IDD could not find a version passing the given test. Unfortunately, revisions 200 and older could not be checked out with the given subversion (`svn`) client. We assume

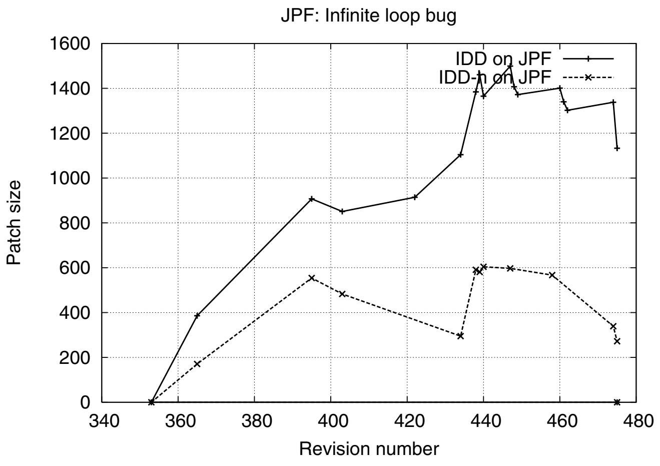


Fig. 9. Result of using IDD on Java PathFinder

that the server-side repository was converted to a later version at that stage, making it impossible to retrieve old revisions.

6.2 Java PathFinder

Java PathFinder (JPF) is a Java model checker that analyzes concurrent Java programs for assertion violations, deadlocks, and other failures [12]. In version 4, a complex input is not processed correctly. The defect is manifest in an incorrect result after several minutes of processing: The analysis report of JPF states that a faulty input (a Java program containing a known defect) is correct. Because the point of failure is extremely difficult to locate, this problem has persisted for more than a year and has not been resolved yet. In fact, it was this case study that triggered the work of this paper.

A much older version of JPF, 3.0a, which is not maintained in the same repository, produces a correct result. However, using that version for identifying a change set would not be useful, because the entire architecture of JPF has been redesigned since version 3.0a was released. Hence, IDD was applied to different revisions of the source repository containing all revisions of version 4. The goal was to find a revision in the new repository that could pass the test. Perhaps the bug was introduced after the architectural changes that took place prior to the migration to a public source repository. In that case, IDD could find it.

We have applied IDD using revision 475 of version 4 as a starting point. IDD iterated through older revisions up to version 353, which passed the test (see Figure 9). In this case, no changes had to be back-ported to that revision. Therefore, the visible part of the graph shows the size of the patch fixing the defect. Delta debugging was unable to identify a very small change set to fix the newer, defective versions. As a result of this, the initial patch of 386 and 171 lines, respectively, grew successively during back-porting. Sometimes, a later iteration of DD took advantage of previously resolved dependencies

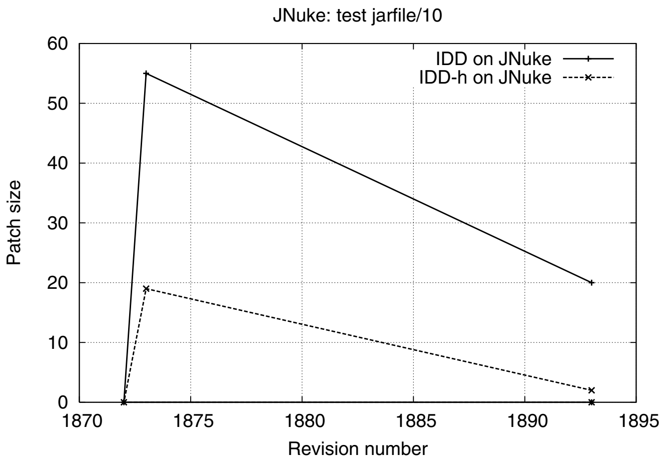


Fig. 10. Result of using IDD on JNuke

and shrank the patch again. However, at the end, an unwieldy patch of more than 1,000 lines remained for standard IDD, and a patch of 272 lines for hierarchical IDD.

This is still too large for the patch being meaningful to a human programmer. The generated change set still includes refactorings amidst functional changes, and while it fixes the given bug, it breaks other features of JPF. Because of this, we did not follow up with this bug fix further to actually repair JPF.

6.3 JNuke

In JNuke, IDD was applied to find the reason of a memory access problem in the Jar file reader under Mac OS X 10.4 that was not found under Linux. This is maybe the most typical case in which IDD can be applied: Code that is often tested on one platform (in this case, Linux) but rarely on another one (Mac OS). Such tests may pass on the main development and test system, but fail on a different platform. As the system is periodically tested on the other platforms, some known good versions exist, but regression defects may go undetected for some time.

As Figure 10 shows, IDD was very successful here. The test could be run without any adaptations on older versions, until it passed at revision 1872. The graph shows the size of the patch fixing the problem. Hierarchical IDD even found a minimal patch of just two lines, identifying the fault precisely. This example is a case where IDD would have found the test quickly enough to be useful for replacing human effort in debugging.

6.4 Summary

Table 2 summarizes the outcome of all experiments. As can be seen, the hierarchical version of IDD always produces significantly smaller patches, although the difference varies. In either case, the number of DD cycles within DD would have been prohibitively large in practice in all cases except for the last one. The complexity of DD

Table 2. Summary of experiments

Project name	Size of final patch set		Number of DD invocations		Number of build/ test runs	
	IDD	IDD-h	IDD	IDD-h	IDD	IDD-h
Uncrustify	666	567	62	57	52483	39170
Java PathFinder	1133	272	18	13	57140	15013
JNuke	20	2	4	4	335	117

is linear in the size of failed build/test runs [14]. However, in IDD, larger patches tend to be more vulnerable to patch conflicts in later revisions. Therefore, once a patch has accumulated a number of features such as refactorings, name changes, etc., the `patch` command tends to fail more often. As a result of this, the DD subroutine is used more often, which tends to lead to even larger patches.

Therefore, when patches get larger, both the size and the frequency of DD runs increases. This means that the number of test/build runs grows faster than linearly in the number of DD invocations. As each build/test cycle takes about 30 seconds in average, longer IDD runs can take several days.[4] In practice, the likelihood of a success is highest for quickly detected defects, so DD would typically be run for a few hours or days before it would be aborted.

7 Conclusion

Delta debugging automates the task of identifying minimal change sets. However, it requires a correct version, which may not be known when a new defect is discovered. Still, it is possible that an old revision passes a given test. Sometimes it is necessary to apply changes of newer versions to older ones in order to allow a newer test to execute on an old version. Iterative delta debugging automates this process and successively carries changes from newer versions back to older ones, until either a correct version is found or the process is aborted. The process is successful in some cases, but requires much time when change sets get large.

8 Future Work

IDD works in conjunction with delta debugging, but requires a high-performance, high-precision implementation of DD in order to be successful.

The current tool chain has been written as a prototype, lacking several possible optimizations that could be implemented. For instance, DD generates and compiles variations of one revision by checking out a fresh copy of the necessary files and compiling them from scratch in each iteration. Whenever the build configuration does not change, most of these steps can be cached.

[4] Intelligent caching of compiled artifacts could reduce this time by a factor of 10. However, we did not find a system that supported all the programming languages and revision control systems in question.

Furthermore, code analysis could eliminate the need to check variations where too much code is removed. Code coverage and dynamic slicing on a known "good" version can help to identify which statements are crucial for a test to pass. Such statements should not be removed in the DD process. For C programs, memory checking algorithms can further automate the suppression of programs that do not generate consistent results.

The direction of applying hierarchical DD is definitely promising, and delivers faster and better results than standard DD. However, the hierarchy of the patch file structure does not exactly mirror the hierarchy of software source code. Tools that represent software source code in XML format [2,6] could be used to extract a hierarchical representation of programming constructs. If XML data is used, then the question of how to generate an efficient and expedient difference representation is still open. Tools having their own format exist [4], and may be used in further case studies.

References

1. Artho, C., Shibayama, E., Honiden, S.: Iterative delta debugging. In: 19th IFIP Int. Conf. on Testing of Communicating Systems (TESTCOM 2007), Tallinn, Estonia (2007)
2. Gondow, K., Suzuki, T., Kawashima, H.: Binary-level lightweight data integration to develop program understanding tools for embedded software in C. In: Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 2004), Washington, USA, 2004, pp. 336–345. IEEE Computer Society Press, Los Alamitos (2004)
3. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. 29(3), 155–163 (1988)
4. Lindholm, T., Kangasharju, J., Tarkoma, S.: Fast and simple XML tree differencing by sequence alignment. In: Proc. 2006 ACM symposium on Document engineering (DocEng 2006), pp. 75–84. ACM, New York (2006)
5. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: PSE: Explaining Program Failures via Postmortem Static Analysis. In: Proc. 12th Int. Symposium on the Foundations of Software Engineering (FSE 2004), pp. 63–72. ACM, New York (2004)
6. Maruyama, K., Yamamoto, S.: A tool platform using an XML representation of source code information. IEICE – Trans. Inf. Syst. E89-D(7), 2214–2222 (2006)
7. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: Proc. 28th Int. Conf. on Software Engineering (ICSE 2006), Shanghai, China, pp. 142–151. ACM Press, New York (2006)
8. Myers, G.: Art of Software Testing. John Wiley & Sons, Chichester (1979)
9. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: Proc. 3rd Int. Workshop on Run-time Verification (RV 2003), Boulder, USA. ENTCS, vol. 89, pp. 22–43. Elsevier, Amsterdam (2003)
10. Peled, D.: Software Reliability Methods. Springer, Heidelberg (2001)
11. Torvalds, L., Hamano, C.: git-bisect (1) manual page (2007), `http://kernel.org/pub/software/scm/git/docs/git-bisect.html`
12. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal 10(2), 203–232 (2003)
13. Weiser, M.: Programmers use slices when debugging. Communications of the ACM 25(7), 446–452 (1982)
14. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. Software Engineering 28(2), 183–200 (2002)

Linear-Time Reductions of Resolution Proofs

Omer Bar-Ilan[1], Oded Fuhrmann[1], Shlomo Hoory[1], Ohad Shacham[1], and Ofer Strichman[2]

[1] IBM Haifa Research Laboratory
{barilan,Odedf,shlomoh,ohads}@il.ibm.com
[2] Information Systems Engineering, IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. DPLL-based SAT solvers progress by implicitly applying binary resolution. The resolution proofs that they generate are used, after the SAT solver's run has terminated, for various purposes. Most notable uses in formal verification are: extracting an *unsatisfiable core*, extracting an *interpolant*, and detecting clauses that can be reused in an *incremental satisfiability* setting (the latter uses the proof only implicitly, during the run of the SAT solver). Making the resolution proof smaller can benefit all of these goals. We suggest two methods that are linear in the size of the proof for doing so. Our first technique, called Recycle-Units, uses each learned constant (unit clause) (x) for simplifying resolution steps in which x was the pivot, prior to when it was learned. Our second technique, called Recycle-Pivots, simplifies proofs in which there are several nodes in the resolution graph, one of which dominates the others, that correspond to the same pivot. Our experiments with industrial instances show that these simplifications reduce the core by $\approx 5\%$ and the proof by $\approx 13\%$. It reduces the core less than competing methods such as run-till-fix, but whereas our algorithms are linear in the size of the proof, the latter and other competing techniques are all exponential as they are based on SAT runs. If we consider the size of the proof graph as being polynomial in the number of variables (it is not necessarily the case in general), this gives our method an exponential time reduction comparing to existing tools for small core extraction. Our experiments show that this result is evident in practice more so for the second method: rarely it takes more than a few seconds, even when competing tools time out, and hence it can be used as a cheap proof post-processing procedure.

1 Introduction

DPLL-based SAT solving became in the last few years the single most-used back-end engine for model checking and satisfiability modulo theories. While SAT solvers exist from at least the 1960's, and learning through derivation of conflict clauses exists from at least the 1980's (while recognized as implicitly derived by resolution), only in 2003 the question of how to produce a resolution proof from a run of a DPLL solver was addressed in practice [18] and implemented. Most modern solvers nowadays are capable of producing such proofs. Further, there are now decision heuristics that are based on an understanding of the DPLL process as a resolution-based proof engine rather than as a search engine. Examples are Ryan's thesis [15] and his SAT solver Siege, which bias conflict-clause generation towards those that will more likely lead to other resolutions, and the work by Gershman et al. [7] on a model for explaining and designing decision heuristics based on an understanding of the SAT solving process as a resolution engine.

The resolution proofs that modern SAT solvers generate are used in a broad range of applications in formal verification and elsewhere. Prominent examples are:

– The resolution proof can be used for extracting an *unsatisfiable core*, which can then be used, e.g., in a proof-based abstraction-refinement procedure, as shown by Amla and McMillan [1]. The unsatisfiable core was also used in the past for detecting the reasons for unsatisfiability in an underapproximation/refinement process for model-checking [9]. In the context of satisfiability modulo theories (SMT): the core is used for finding small *explanations* (see, for example, the recent work by Cimatti et al. [4], who suggest to invoke a tool for minimizing resolution proofs for this purpose) and in theory-specific decision procedures, such as the abstraction-based procedures for Presburger and bitvector formulas proposed by Kroening et al. and Bryant et al., respectively [3,10].
– The resolution proof can be used for extracting an *interpolant* as part of a complete model-checking technique, as suggested by McMillan in [12].
– The proof can be used for detecting clauses that can be reused in an *incremental satisfiability* setting [16], such as the one used in Bounded Model-Checking. The analysis of the proof is done implicitly, during the run of the SAT solver (in contrast to the first two uses), and is required in order to check whether all the clauses that were used for resolving a particular conflict clause are still expected to be present in the new SAT instance. If yes, this conflict clause can be reused.

Making the proof smaller by removing some of the nodes and removing literals from other nodes can benefit all of these goals. In all of these applications, however, reducing the size of the core/interpolant/reused-clauses has an influence on the overall run time which is somewhat unpredictable and only vaguely known. Further, the proof reduction component is called many times during the overall solving process. As a result, best overall results are most likely to be achieved with a limited investment in such reductions.

The script Run-till-fix by Zhang and Malik [18] extracts a core and attempts to minimize it by simply running the SAT solver on it repeatedly until reaching a fix-point. Achieving fast reductions with this tool is not always possible, as it requires a normal SAT run. The goal of achieving fast reductions was first addressed in the work by Gershman et al. [6], based on analyzing the proof graph and trying to restructure it with the aid of a SAT solver. The tasks the SAT solver is asked to solve by their script Trim-till-fix are closely related, and hence incremental satisfiability makes this process relatively fast. There are also many published techniques for finding *minimal* cores, i.e., an unsatisfiable subset of clauses from which no clause can be removed without making it satisfiable (this is also known in the literature by the name MUS, for Minimal Unsatisfiable Subformula). Some works in this direction from the last three years are [13,5,8], all of which are worst-case exponential. The complexity of the decision problem corresponding to finding the minimal unsatisfiable core is D^P-complete[1] [14]. A minimal core (in contrast to the *minimum* core) is not unique and depends on the starting point, like all the methods we are aware of (including the current one). Therefore whether the core found is minimal or just 'small', as indicated in [6], has little significance in practice.

The proof reductions techniques we suggest here are linear in the size of the proof graph. The proof graph itself can be exponential in the number of variables, and hence in the worst case our procedure is still exponential in the number of variables, like Run-till-fix. However, if we assume that in practice the size of the proof graph is not more than the number of variables multiplied by some polynomial with a bounded exponent, there is a gap of an exponent between our method and the competing tools, as they can still be exponential in the number of variables regardless of the size of the proof. Our techniques typically reduce the core significantly less than the exponential methods, but do so much faster. They also reduce the size of the proof itself whereas a procedure like run-till-fix increases it. We therefore consider them as useful tools in the context of short time-outs and where the size of the proof matters.

The first method we suggest is called Recycle-Units. The idea is to remove edges from the proof graph by using information that is inferred by the SAT solver only *after* the resolutions at the nodes adjacent to these branches were made originally. For example, if (x) is a unit clause that was learned by the SAT solver, it can be used for simplifying resolution inferences that used x as the pivot prior to learning this clause. If not carefully applied, however, this may lead to circular reasoning.

The second method is called Recycle-Pivots. It is based on the following observation. For simplicity assume that the proof graph is a tree. Let n_1 and n_2 be two nodes on the same path in the proof tree such that n_1 is closer to the root. Further, assume that the pivot variable associated with both nodes is the same. Our convention is that proofs progress from top to bottom, from the premises (also called the *axioms*) of the proof to its consequent. We also follow the convention by which the right parent of each node contains the negative phase of the pivot variable, and the left parent contains the positive phase of this variable. Assume that n_2 is on the right branch of n_1. In this case, we will show, the left incoming branch of n_2 can be pruned, and the proof rewritten without it in a way that the resulting proof is a legal resolution proof with a smaller core. Since in practice proof graphs are DAGs and not just trees, the procedure is somewhat more complicated as we later describe.

1 D^P is the class containing all languages that can be considered as the difference between two languages in NP, or equivalently, the intersection of a language in NP with a language in co-NP.

The two techniques tighten the proof, which means that the resulting proof uses a subset of the core used by the original proof, and there is an injective mapping between the target and source nodes, such that each target node is either equal or subsumes the source node to which it is mapped. As a theoretical curiosity, we note that the resolution proofs generated by our techniques cannot necessarily be generated by any modern SAT solver and hence by any core reduction technique that is based on rerunning such a solver. Our technical report gives more details about this issue [2].

The rest of the paper is structured as follows: Section 2 summarizes some preliminaries necessary for the description of the technique (we assume, however, that the reader is familiar with SAT basics). Section 3 and 4 describe the two techniques by giving pseudo code and intuitive explanation of their correctness. Section 5 is dedicated to a formal proof of their correctness. We conclude in Section 6 with a description of the experiments we conducted and their results.

2 Preliminaries

A literal is a Boolean variable or its negation. A pair of literals corresponding to a variable and its negation are called *complementary*. A clause is a (possibly empty) disjunction of literals, and a CNF formula is a conjunction of clauses.

2.1 Inference by Resolution

The process in which DPLL SAT solvers infer new conflict clauses can be interpreted as applying the *binary resolution* inference rule:

$$\frac{(l \vee l_1 \vee .. \vee l_n) \qquad (\neg l \vee l'_1 \vee .. \vee l'_n)}{(l_1 \vee .. \vee l_n \vee l'_1 \vee .. \vee l'_n)} \quad (\textsc{Resolution}). \tag{1}$$

The variable l is called the *pivot variable* (also called 'resolution variable' in the literature).

Let *Res* be a function that receives two clauses with complementary literals as input, and returns the consequent of the resolution rule applied to these two clauses, as output. More formally:

Given clauses $C_1 = (l \vee l_1 \vee \; .. \; \vee l_n)$ and $C_2 = (\neg l \vee l'_1 \vee l'_2 \vee \; .. \; \vee l'_n)$,

$$Res(C_1, C_2) = (l_1 \vee \; .. \; \vee l_n \vee l'_1 \vee \; .. \; \vee l'_n) \; .$$

Resolution is known to be a sound and complete proof system for CNF formulas. Specifically, a CNF formula φ is unsatisfiable if and only if there exists a resolution proof of the empty clause using φ's clauses as premises.

Definition 1 (Resolution Graph). *A resolution graph corresponding to a resolution proof is a directed acyclic graph (DAG), where the nodes represent clauses* *and for every pair of nodes C_i, C_j, (C_i, C_j) is an edge if and only if C_i is an antecedent of C_j in the resolution proof.*[2]

Example of a resolution graph can be seen in Fig. 1[a]. Every resolution proof can be represented by a resolution graph. If a resolution graph has a single sink, it is called the *consequent* of the proof. The root nodes are the premises of the proof. For a given node, the root nodes that can reach it on the proof graph are called its *core*. Specifically, if the consequent is an empty clause then its core is called the *unsatisfiable core*.

The resolution described so far is known by the name *general resolution*. Two well-known restrictions of general resolutions that we will mention later on are *tree-like resolution*, which means that the proof graph corresponding to the proof is a tree rather than a DAG, and Tseitin's *regular resolution* [17], which means that along each path no variable is used twice as a pivot. Both of these restrictions may cost in a penalty of an exponent in the size of the proof. In other words, there are formulas that can be proven with general resolution in a polynomial number of steps, but only with an exponential number of tree-like resolution or regular resolution.

How do resolution proofs relate to proofs of SAT solvers? Modern DPLL SAT solvers generate *conflict clauses* during their run, which are implicitly inferred from other clauses by a chain of (general) resolutions. Hence, a proof of unsatisfiability given by solvers have original clauses as their roots, and conflict clauses as their internal nodes. We refer the reader to [18] for more details.

We are going to use a variant of the resolution graph in which a single parent is allowed, if this parent is associated with the same clause as the child. The resolution proof corresponding to this graph is derived by first eliminating all nodes with a single parent (removing such a node n and connecting n's single parent to n's children), and continuing as before. This extension is convenient for simplifying the algorithm and later on the proofs, but is not essential.

3 Recycling Learned Unit Clauses

Some of the conflict clauses learned during the run of a SAT solver are unit clauses, e.g., (x). In such a case we say that the SAT solver inferred the *constant value* of the variable constrained by this clause ($x = true$ in this case). These constants can be used to rewrite the proof starting from those parts of the proof that were inferred prior to learning these constants, an action that reduces the overall size of the proof and its core. In other words, the SAT solver can only apply resolution to clauses in its clause database (which, recall, is continuously updated with new conflict clauses). Further, it can only use resolution variables which at that point in time are unassigned. If at a later stage of the computation

[2] Note that we use the convention by which the edges are in the direction of the proof, i.e., from premises to consequent. For practical purposes it is common to build this graph with edges (pointers) pointing in the other direction, because this facilitates a search for the core.

the same resolution variable is proved to have a constant value then the proof can be regenerated taking this information into account.

Algorithm 1 presents RECYCLE-UNITS, which is the first of a two-step algorithm for recycling unit clauses. The second step is RECONSTRUCT-PROOF, which is presented in Algorithm 2. The two algorithms use the following notation. P is a resolution proof of the empty clause, and $P.sink$ is the node representing the empty clause. For a given node n in P, $n.C$ is the clause represented by n, $n.L$ and $n.R$ are the left and right parents of n respectively, and $n.piv$ is the pivot variable used to resolve $n.C$ from $n.L.C$ and $n.R.C$. Recall that we use the convention by which the left parent includes the positive phase of the pivot, and the right parent includes its negative phase. If l is a literal, we denote by $var(l)$ its corresponding variable. When $n.C$ is a unit clause, we sometimes refer to it as a literal rather than a clause, when the meaning is clear from the context. For example $var(n.C)$ is the variable corresponding to the literal in the unit clause $n.C$.

Algorithm 1. RECYCLE-UNITS(PROOF P)

```
1: Let $U$ be the set of nodes representing constants proved in $P$;
2: for each $u \in U$ do
3:     Mark the (recursive) antecedents of $u$;
4:     for each unmarked $n \in P$ do
5:         if $n.piv == u.C$ then
6:             $n.L = u$;
7:         else if $n.piv == \neg u.C$ then
8:             $n.R = u$;
```

RECYCLE-UNITS iterates over all constants that were proved in P. Let u be a node representing such a constant, and assume for now that this constant is a positive literal (i.e., $u.C$ is a positive literal). First, in line 3, RECYCLE-UNITS marks the nodes in its antecedents closure, i.e., the nodes that can reach U in P. Then, it searches unmarked nodes for those that represent a resolution step using $var(u.C)$ as pivot. Let n be such a node. According to our convention $u.C \in n.L.C$ and $\neg u.C \in n.R.C$. In line 6 RECYCLE-UNITS replaces the left parent of n from $n.L$ to u, i.e., it disconnects the edge $(n.L, n)$ and connects instead (u, n). If $u.C$ is a negative literal, then the edge is shifted from $(n.R, n)$ to (u, n) in line 8.

At this stage the proof is no longer a legal resolution proof, and hence a reconstruction phase begins by calling RECONSTRUCT-PROOF, which appears in Alg. 2.

Example 1. Consider the partial proof graph that is depicted in Fig. 1a. The unit clause C8 is proven only after the resolution of C3, which uses the variable '1' (the unit of C8) as its pivot. RECYCLE-UNITS begins by marking clauses that were used for proving C8 – clauses C5 and C6 in this case. It then identifies C3 as an unmarked node that uses '1' as pivot and rewires the proof – in this

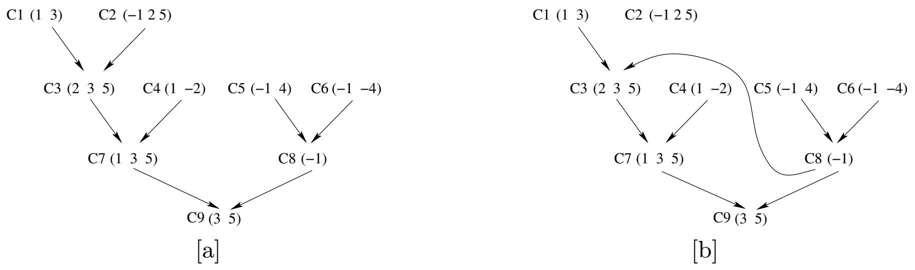


Fig. 1. [a] Part of a resolution proof [b] After Recycle-Units and before Reconstruct-Proof

case disconnects (C2,C3) and adds instead the edge (C8,C3), as can be seen in Fig. 1b. It is left to reconstruct the proof so it becomes a legitimate resolution proof once again. □

Implementation of Recycle-Units*.* The most time-consuming component of Algorithm 1 is Step 3. Recall that the purpose of this step is to prevent cycles after we connect a unit clause to another node. Rather than traversing the graph backwards each time (which would make this method quadratic in the size of the graph), our implementation maintains at each unit node pointers to all immediate descendent units. Denote by G_U the resulting graph of units, i.e., the nodes of G_U are the units in the resolve graph and the edges are defined by the list of pointers that we maintain with each such node. In Step 3, when considering connecting a unit clause u to another node c, our tool temporarily makes this connection and checks if it can reach itself on G_U. If the answer is yes, we undo this temporary connection and continue. Otherwise we keep the connection and update the list of pointers in the units that are immediate antecedents of u so they now also point to u. This method makes the solution quadratic in the number of units. We can always bound the number of units that we consider (regardless – in practice it is small comparing to the size of the proof) and hence refer to this element in the complexity analysis as a constant.

Let us now shift the focus to Reconstruct-Proof. The proof graph given to Reconstruct-Proof as input has a subset of the nodes of the original proof graph. This is because only edges are shifted up to this point, and these shifts may disconnect parts of the graph (e.g., in Example 1, node C2 has been disconnected). Note that only the graph connected to the sink node is sent to Reconstruct-Proof.

Reconstruct-Proof, appearing in Alg. 2, is a procedure that, given such a "broken" proof P and a node n (initially the empty clause), reconstructs a legal resolution proof of n (we will define formally the nature of this broken proof in Section 5). The procedure is recursive starting from the node n. The base of the recursion are the root nodes. When n is not a leaf, there are several cases. If the pivot $n.piv$ is still present in its parents then $n.C$ is a resolution between them (see lines 10 and 11). If it is only present in one of them, say the positive phase is present in $n.L.C$, then the other node ($n.R$) replaces n. This is because we know that $n.R.C$ subsumes $n.C$. If it is contained in neither of its parents, then both $n.L.C$ and $n.R.C$ subsume $n.C$ and hence either one of them can replace $n.C$.

Algorithm 2. Reconstruct-Proof (“Broken” proof P, node n)

```
1: if $n$ visited then return
2: mark $n$ as visited;
3: if $n$ is a root then return
4: if $n$ has a single parent $n.L$ then
5:     Reconstruct-Proof ($P$,$n.L$);
6:     $n.C = n.L.C$;
7: else
8:     Reconstruct-Proof ($P$,$n.L$);
9:     Reconstruct-Proof ($P$,$n.R$);
10: if $n.piv \in n.L.C$ and $\neg n.piv \in n.R.C$ then
11:     $n.C = Res(n.L.C, n.R.C)$;
12: else if $n.piv \in n.L.C$ and $\neg n.piv \notin n.R.C$ then
13:     $n.C = n.R.C$;
14:     $n.L = $ nil
15: else if $n.piv \notin n.L.C$ and $\neg n.piv \in n.R.C$ then
16:     $n.C = n.L.C$;
17:     $n.R = $ nil;
18: else
19:     $side$ = one of $\{L, R\}$; $otherside$ = other side;    ▷ Choose heuristically
20:     $n.C = n.side.C$;
21:     $n.otherside = $ nil;
```

Example 2. The graphs in Fig. 2 show the steps of reconstruction for the proof graph in Fig. 1b. □

4 Recycling Pivots

The second technique we present is linear in the size of the proof and is also based on two steps, as in the previous case: in the first step we remove edges from the proof, and in the second step we reconstruct the proof using the same algorithm Reconstruct-Proof. The algorithm is based on the observation that along each path from root to sink, there is no need for resolving on the same variable more than once. If there is such a situation, then the redundant resolution steps can be avoided and consequently some of the branches of the proof can be pruned away. The correctness of this algorithm will be proven in Section 5. We note that this does not result in a regular resolution proof as defined in Section 2, because we apply it only to some parts of the graph as described below.

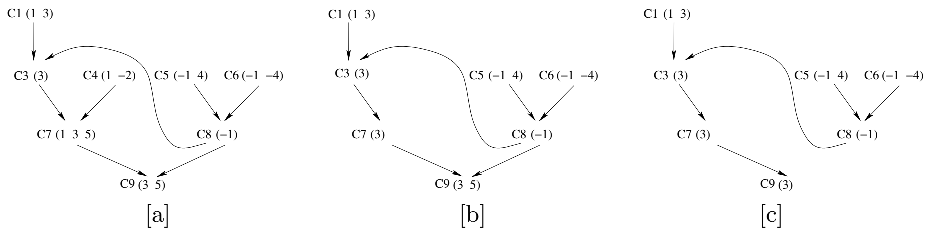


Fig. 2. The recursive steps, from left to right, of reconstructing the proof graph in Fig. 1b. Recall that single parent nodes are consistent with our definition of resolution graphs.

If the resolution graph we start with happens to be a tree-like resolution, then the result of applying this step is indeed a regular resolution proof.

For simplicity of the presentation, assume for now that the proof is a tree rather than a DAG. Consider a node n with a pivot $n.piv$. We propagate $n.piv$ up the right branch (recall that according to our convention $n.R.C$ contains $\neg n.piv$) in a set called *Removable-Literals*, or RL for short. Similarly, we propagate $\neg n.piv$ up the left branch. Consider the right branch: if along this branch there is another node n' such that $n'.piv = n.piv$ then we can replace n' with $n'.R$. This means that the branch starting at $n'.L$ is pruned. The correctness of this operation is tied to the second step, which is the proof reconstruction in Reconstruct-Proof. Note that $n'.R.C$ is contained in $n'.C$ other than $\neg n.piv$. Reconstruct-Proof will effectively propagate $\neg n.piv$ down the branch until inevitably reaching n (a result of our assumption that this is a tree). At node n, $\neg n.piv$ will disappear again due to the resolution on $n.piv$ at n. As a result $n.C$ will subsume its original version.

In practice the input proof can be a DAG. This may cause a situation in which our node n' has paths to the sink not through n. This, in turn, may cause a situation that Reconstruct-Proof propagates $\neg n.piv$ all the way down to the sink, which contradicts our goal of producing a proof with an equal or stronger consequent. Another possible problem is that n' has paths to the sink node through both incoming edges of n, which nullifies our suggested technique. There are two possible solutions to this problem. One, which is the solution taken in the pseudo-code described here (see line 4) and also in our implementation, is to propagate RL up (towards the roots) only as long as it is a tree. A more complicated solution, which we leave for future work, is to check whether all paths from n' to the sink go through the edge $(n.R, n)$. This can be done by computing *dominance* relation in the graph, for example with the Lengauer-Tarjan algorithm[11] (which runs in $O(|E| \log |V|)$ time). It might burden the computation, however, since the dominance relation has to be recomputed after each removal of an edge.

Example 3. Assume Fig. 3a represents the input proof for Recycle-Pivots. Recycle-Pivots propagates up the removable literals (denoted by RL in the

Algorithm 3. Recycle-Pivots(n, RL)

```
1: if $n$ visited then return
2: Mark $n$ as visited
3: if leaf then return
4: if $n$ has more than one child then RL = {}
5: if $piv \notin$ RL and $\neg piv \notin$ RL then
6:     Recycle-Pivots($n.L$, RL $\cup \{\neg piv\}$)
7:     Recycle-Pivots($n.R$, RL $\cup \{piv\}$)
8: else if $piv \in$ RL then                    ▷ this implies $\neg piv \notin$ RL
9:     $n.L$ = nil;
10:    Recycle-Pivots($n.R$, RL);
11: else                                        ▷ $piv \notin$ RL and $\neg piv \in$ RL
12:    $n.R$ = nil;
13:    Recycle-Pivots($n.L$, RL);
```

drawing) and, owing to the fact that '2' is the pivot of C3 and that '2' is in RL, erases the edge (C1,C3). The proof after calling Reconstruct-Proof is depicted in Fig. 3b. □

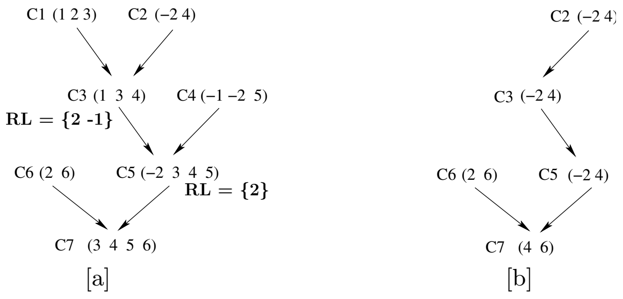


Fig. 3. For the input proof presented in drawing [a], Recycle-Pivots erases the edge (C1,C3) and then calls Reconstruct-Proof, which results in the graph in drawing [b]. **RL** represents the Removable-Literals sets.

5 Proofs

The proof of correctness relies on a notion of *e-resolution*, which we soon define. As we will prove, the graphs produced by Recycle-Units and Recycle-Pivots are e-resolution proof graphs, and Reconstruct-Proof transforms them back to resolution graphs.

It is convenient to represent resolution proofs as a DAG in which only the roots are labeled with clauses. The clauses labeling the other nodes, including the consequents, can be inferred from the topology of the graph and the roots, and hence are not considered as part of the representation. Recall that single parents are allowed if the parent is labeled with the same clause as the child.

e-resolution is defined based on this convention, with the difference that each internal node n with two parents is labeled with a variable $n.piv$,[3] which in our case is the pivot used to infer $n.C$ at that node in the original proof. For nodes with two parents, instead of Eq. (1), e-resolution uses a more relaxed inference rule, of which (1) is a special case. Using the convention that n.piv may occur only in $n.L.C$, and $\neg n$.piv may occur only in $n.R.C$, the e-resolution inference rule is:

$$\begin{aligned} n.C &= \text{e-resolution}_{n.\text{piv}}(n.L.C, n.R.C) \\ &= (n.L.C \setminus \{n.piv\}) \cup (n.R.C \setminus \{\neg n.piv\})\ . \end{aligned} \tag{2}$$

For example, for a node n labeled with a pivot x, and parents $n.L.C = (y_1 \vee y_2)$ and $n.R.C = (\neg x \vee y_3)$ we have $n.C = (y_1 \vee y_2 \vee y_3)$.

5.1 The Reconstruct-Proof Algorithm

In the following we denote by $\leq$ the subsumption relation, e.g., $C_1 \leq C_2$ means that C_1 has a subset of the literals of C_2. By $A(P)$ we denote the set of assumptions (roots) of a proof P.

Lemma 1. *Let P be an e-resolution proof of C, and let P' =*Reconstruct-Proof*(P). Then P' is a resolution proof of C', where $C' \leq C$ and $A(P') \subseteq A(P)$.*

Proof. The claim follows by induction on the height of the proof (the length of the longest path from a root to the sink). The proof is a straight-forward analysis of the four possible cases for a node n as described by the table in Fig. 4. Since by the induction hypothesis we can assume that the proofs of $n.L.C$ and $n.R.C$ are legitimate resolution proofs, then clearly the second and third cases, which simply copy a parent node, result in a resolution proof of $n.C$. In the fourth case the choice is made heuristically to achieve best pruning, and the induction step applies to both choices. It is also clear that in all cases, at each step the clause labeling n can only be smaller than the one required by applying the e-resolution inference rule (2) to the updated $n.L.C$ and $n.R.C$. This of course applies to the consequent clause as well.

5.2 The Recycle-Units Algorithm

Recycle-Units is a special case of a more general procedure, which we call **subsumption**. It generalizes Recycle-Units in the sense that it does not only recycle unit clauses, rather any learnt clause that subsumes other clauses in the proof. Our proof will refer to subsumption, and the correctness of Recycle-Units is then implied.

[3] This is in contrast to a standard resolution graph, in which the pivots can be inferred from the topology and axioms.

$n.C$	$n.\text{piv} \in n.L.C$	$\neg n.\text{piv} \in n.R.C$
$\text{Res}(n.L.C, n.R.C)$	yes	yes
$n.L.C$	no	yes
$n.R.C$	yes	no
$n.R.C$ or $n.L.C$	no	no

Fig. 4. The four cases discussed in the proof of Lemma 1

Consider two nodes p, m in the proof P such that $p.C \leq m.s.C$ for some side $s \in \{L, R\}$. Then, as long as no cycle is produced, we can set $m.s = p$ so that m uses the stronger p instead of its original parent $m.s$. In fact, the next definition and lemma show that one can perform more than one such subsumption operation in parallel, and the definition of subsumption can be slightly strengthened by considering the pivot variable. Indeed, Recycle-Units as listed in Algorithm 1 performs such substitutions with an arbitrary order.

Definition 2 (e-subsumption). *Let P be an e-resolution proof. We say that the node p e-subsumes the k nodes $m_1, \ldots, m_k$ with sides $s_1, \ldots, s_k$ (i.e., $s_i \in \{L, R\}$), if the following conditions hold for every i: The node m_i has two parents; m_i is not an ancestor of p; and*

$$p.C \leq \begin{cases} m_i.L.C \cup \{m_i.piv\} & \textit{if } s_i = L \\ m_i.R.C \cup \{\neg m_i.piv\} & \textit{if } s_i = R \,. \end{cases}$$

Lemma 2 (Parallel e-subsumption). *Let P be an e-resolution unsatisfiability proof, and let $p, m_1, \ldots, m_k$, $s_1, \ldots, s_k$ be as above. Then the proof P' obtained by setting $m_i.s_i = p$ for all i is an e-resolution unsatisfiability proof with $A(P') \subseteq A(P)$.*

Proof. Let P, P' be as above. We first argue that P' contains no cycles. Indeed, assume such a cycle exists. Obviously the cycle must contain some new edge. However, since all the new edges emanate from p, we can assume that the cycle contains exactly one new edge, say the edge from p to m_1. However, the rest of the cycle is a path from m_1 to p, which is a contradiction to the assumption that m_1 is not an ancestor of p.

Second, we argue that P' is an e-resolution unsatisfiability proof. The proof is by induction on the height of n with the induction claim: $n.C' \leq n.C$. Here we let $n.C, n.C'$ denote the consequent clauses at the node n when applying the e-resolution inference rule for the proof P, P' respectively. As the claim trivially holds for root nodes and for nodes with in-degree one, we can restrict our attention to some node n with two parents. By induction $n.L.C' \leq n.L.C \cup \{n.\text{piv}\}$ and $n.R.C' \leq n.R.C \cup \{\neg n.\text{piv}\}$.[4] Therefore, by (2) we conclude that $n.C' \leq n.C$ as claimed.

(Proof of the correctness of Recycle-Units)

[4] Note that the stronger claim $n.L.C' \leq n.L.C$ may not be correct since $n.\text{piv}$ may have been added to $n.L.C$ if $n.L$ is one of the k nodes modified in the transition from P to P'.

Proof. The claim follows by applying Lemma 2 once for every iteration starting in line 2 of the Recycle-Units algorithm, and subsequently applying Lemma 1 once.

The only point that needs some extra elaboration is that the assumptions of Lemma 2 hold each time it is applied. Indeed, the algorithm Recycle-Units only considers the case of subsumption by a unit clause u. Furthermore, the algorithm checks for the condition $u = n.\text{piv}$ if $s_i = L$ and $u = \neg n.\text{piv}$ if $s_i = R$, which is strictly stronger than required by the Lemma.

The proof of correctness of Recycle-Pivots is given in the full version of this article [2] due to lack of space.

6 Experimental Results and Conclusions

We ran our linear reductions on the IBM benchmark suite, which comprises 63 different designs. On each design we ran bounded model checking with a bound k initially set to 10 and then incremented by 5, up to $k = 100$ or a bug was found. This gave us 630 unsatisfiable instances. From those we chose only the instances that take 10 seconds or more for run-till-fix.[5] This left us with 67 proofs. We set the timeout for each run to be 1800 seconds. We used a 64-bit machine with 8 GB memory, and 2x2.4Ghz Opteron dual core.

		Leaves				Nodes			
Reduction	Time	Before	After	Per sec	Ratio	Before	After	per sec	Ratio
run-till-fix	8095	1002924	533941	57.9	0.53	11830898	17677419	-722.2	1.49
units	1002.5	1002924	997674	5.2	0.99	11830898	11513195	316.9	0.97
pivots	32.5	1002924	953585	1518.6	0.95	11830898	10464394	42059.2	0.88
units + pivots	1235.8	1002924	949279	43.4	0.95	11830898	10247401	1281.3	0.87

Fig. 5. Reduction in proof leaves and nodes. Run time is cumulative for 63 unsatisfiable runs. The 'Per sec' columns indicate the number of removed leaves / nodes per second. The 'Ratio' columns indicate the ratio between the number of leafs (or nodes) before and after the reduction.

The results appear in Fig. 5. What can be concluded from them is that the linear reductions we proposed are on one hand fast (especially Recycle-Pivots), but their effectiveness in reducing leaves is small: only $\approx 5\%$ of the leaves are removed, comparing to 47% with run-till-fix. When it comes to the size of the proof itself, it turns out that run-till-fix *increases* the number of proof nodes substantially (by 49%) whereas our linear reductions decrease their size by 13%. The size of the proof can be relevant when computing interpolants, and indeed

[5] This creates a bias against run-till-fix, but recall that we are not competing against run-till-fix – we only check whether our methods can be helpful when run-till-fix fails with a short time-out.

as future work we intend to check its effectiveness on IBM's interpolation-based model-checker.

To conclude, we showed two techniques for fast preprocessing (perhaps it should be called postprocessing) of resolution proofs. They cannot replace RUN-TILL-FIX if the main goal is the reduction in core regardless of the time it takes, but they can complement it or even replace it in a realm of short time outs. As indicated above, it is more valuable in scenarios in which decreasing the proof size rather than its core is what matters.

References

1. Amla, N., McMillan, K.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
2. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs (full version). Technical Report IE/IS-2008-02, Technion, Industrial Engineering (2008)
3. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
4. Cimatti, A., Griggio, A., Sebastiani, R.: A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 334–339. Springer, Heidelberg (2007)
5. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
6. Gershman, R., Koifman, M., Strichman, O.: Deriving small unsatisfiable cores with dominators. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 109–122. Springer, Heidelberg (2006)
7. Gershman, R., Strichman, O.: Haifasat: A new robust SAT solver. In: Wolfsthal, Y., Ur, S., Bin, E. (eds.) HVC 2005. LNCS, vol. 3875, pp. 76–89. Springer, Heidelberg (2006)
8. Grégoire, É., Mazure, B., Piette, C.: Local-search extraction of muses. Constraints 12(3), 325–344 (2007)
9. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 122–131. ACM Press, New York (2005)
10. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic. In: Alur, R., Peled, D. (eds.) CAV 2004. LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004)
11. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
12. McMillan, K.: Interpolation and sat-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
13. Mneimneh, M.N., Lynce, I., Andraus, Z.S., Silva, J.P.M., Sakallah, K.A.: A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 467–474. Springer, Heidelberg (2005)

14. Papadimitriou, C.H., Wolfe, D.: The complexity of facets resolved. J. Comput. Syst. Sci. 37(1), 2–13 (1988)
15. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University (2004)
16. Shtrichman, O.: Prunning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, p. 58. Springer, Heidelberg (2001)
17. Tseitin, G.: On the complexity of proofs in poropositional logics. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, vol. 2. Springer, Heidelberg (1983); Originally published 1970
18. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formulas (2003)

Significant Diagnostic Counterexamples in Probabilistic Model Checking

Miguel E. Andrés[1,⋆], Pedro D'Argenio[2,⋆⋆], and Peter van Rossum[1]

[1] Institute for Computing and Information Sciences, The Netherlands
{mandres,petervr}@cs.ru.nl

[2] FaMAF, Universidad Nacional de Córdoba, CONICET, Argentina
dargenio@famaf.unc.edu.ar

Abstract. This paper presents a novel technique for counterexample generation in probabilistic model checking of Markov chains and Markov Decision Processes. (Finite) paths in counterexamples are grouped together in witnesses that are likely to provide similar debugging information to the user. We list five properties that witnesses should satisfy in order to be useful as debugging aid: similarity, accuracy, originality, significance, and finiteness. Our witnesses contain paths that behave similarly outside strongly connected components.

Then, we show how to compute these witnesses by reducing the problem of generating counterexamples for general properties over Markov Decision Processes, in several steps, to the easy problem of generating counterexamples for reachability properties over acyclic Markov chains.

1 Introduction

Model checking is an automated technique that, given a finite-state model of a system and a property stated in an appropriate logical formalism, systematically checks the validity of this property. Model checking is a general approach and is applied in areas like hardware verification and software engineering.

Nowadays, the interaction geometry of distributed systems and network protocols calls for probabilistic, or more generally, quantitative estimates of, e.g., performance and cost measures. Randomized algorithms are increasingly utilized to achieve high performance at the cost of obtaining correct answers only with high probability. For all this, there is a wide range of models and applications in computer science requiring quantitative analysis. Probabilistic model checking allows to check whether or not a probabilistic property is satisfied in a given model, e.g., "Is every message sent successfully received with probability greater or equal than 0.99?".

A major strength of model checking is the possibility of generating diagnostic information in case the property is violated. This diagnostic information is provided through a *counterexample* showing an execution of the model

⋆ Supported by NWO project 612.000.526.

⋆⋆ Supported by the ANPCyT project PICT 26135 and CONICET project PIP 6391.

that invalidates the property under verification. Besides the immediate feedback in model checking, counterexamples are also used in abstraction-refinement techniques [CGJ+00], and provide the foundations for schedule derivation (see, e.g., [BLR05]).

Although counterexample generation was studied from the very beginning in most model checking techniques, this has not been the case for probabilistic model checking. Only recently [AHL05, AD06, AL06, HK07a, HK07b, AL07] attention was drawn to this subject,fifteen years after the first studies on probabilistic model checking. Contrarily to other model checking techniques, counterexamples in this setting are *not* given by a single execution path. Instead, they are *sets of executions* of the system satisfying a certain undesired property whose probability mass is higher than a given bound. Since counterexamples are used as a diagnostic tool, previous works on counterexamples have presented them as sets of *finite* paths with probability large enough. We refer to these sets as *representative counterexamples*. Elements of representative counterexamples with high probability have been considered the most informative since they contribute mostly to the property refutation.

A challenge in counterexample generation for probabilistic model checking is that (1) representative counterexamples are very large (often infinite), (2) many of its elements have very low probability (which implies that are very distant from the counterexample), and (3) that elements can be extremely similar to each other (consequently providing similar diagnostic information). Even worse, (4) sometimes the finite paths with highest probability do not indicate the most likely violation of the property under consideration.

For example, look at the Markov chain $\mathcal{D}$ in Figure 1. The property $\mathcal{D} \models_{\leq 0.5} \Diamond\psi$ stating that execution reaches a state satisfying ψ (i.e., reaches s_3 or s_4) with probability lower or equal than 0.5 is violated (since the probability of reaching ψ is 1). The left hand side of table in Figure 2 lists finite paths reaching ψ ranked according to their probability. Note that finite paths with highest probability take the left branch in the system, whereas the right branch in itself has higher probability, illustrating Problem 4. To adjust the model so that it does satisfy the property (bug fixing), it is not sufficient to modify the left hand side of the system alone; no matter how one changes the left hand side, the probability of reaching ψ remains at least 0.6. Furthermore, the first six finite paths provide similar diagnostic information: they just make extra loops in s_1. This is an example of Problem 3. Additionally, the probability of every single finite path is far below the bound 0.5, making it unclear if a particular path is important; see Problem 2 above. Finally, the (unique) counterexample for the property $\mathcal{D} \models_{<1} \Diamond\psi$ consists of infinitely many finite paths (namely all finite paths of $\mathcal{D}$); see Problem 1. To overcome these problems, we partition a representative counterexample into sets of finite paths that follow a similar pattern. We call these sets *witnesses*. To ensure that witnesses provide valuable diagnostic information, we desire that the set of witnesses that form a counterexample satisfies several properties: two different witnesses should provide different diagnostic information (solving Problem 3) and elements of a single witness should provide similar

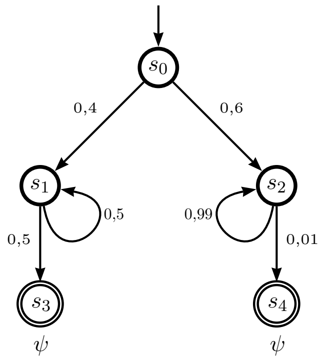


Fig. 1. Markov chain

	Single paths		**Witnesses**	
Rank	F. Path	Prob	Witness	Mass
1	$s_0(s_1)^1s_3$	0.2	$[s_0s_2s_4]$	0.6
2	$s_0(s_1)^2s_3$	0.1	$[s_0s_1s_3]$	0.4
3	$s_0(s_1)^3s_3$	0.05		
4	$s_0(s_1)^4s_3$	0.025		
5	$s_0(s_1)^5s_3$	0.0125		
6	$s_0(s_1)^6s_3$	0.00625		
7	$s_0(s_2)^1s_4$	0.006		
8	$s_0(s_2)^2s_4$	0.0059		
9	$s_0(s_2)^3s_4$	0.0058		
$\vdots$	$\vdots$	$\vdots$		

Fig. 2. Comparison Table

diagnostic information, as a consequence witnesses have a high probability mass (solving Problems 2 and 4), and the number of witnesses of a representative counterexample should be finite (solving Problem 1).

In our setting, witnesses consist of paths that behave the same outside strongly connected components. In the example of Figure 1, there are two witnesses: the set of all finite paths going right, represented by $[s_0s_2s_4]$ whose probability (mass) is 0.6, and the set of all finite paths going left, represented by $[s_0s_1s_3]$ with probability (mass) 0.4.

In this paper, we show how to obtain such sets of witnesses for bounded probabilistic LTL properties on Markov Decision Processes (MDP). In fact, we first show how to reduce this problem to finding witnesses for upper bounded probabilistic reachability properties on discrete time Markov chains (MCs). The major technical matters lie on this last problem to which most of the paper is devoted.

In a nutshell, the process to find witnesses for the violation of $\mathcal{D} \models_{\leq p} \lozenge\psi$, with $\mathcal{D}$ being an MC, is as follows. We first eliminate from the original MC all the "uninteresting" parts. This proceeds as the first steps of the model checking process: make absorbing all states satisfying ψ, and all states that cannot reach ψ, obtaining a new MC $\mathcal{D}_\psi$. Next reduce this last MC to an acyclic MC $\mathrm{Ac}(\mathcal{D}_\psi)$ in which all strongly connected components have been conveniently abstracted with a single probabilistic transition. The original and the acyclic MCs are related by a mapping that, to each finite path in $\mathrm{Ac}(\mathcal{D}_\psi)$ (that we call *rail*), assigns a set of finite paths behaving similarly in $\mathcal{D}$ (that we call *torrent*). This map preserves the probability of reaching ψ and hence relates counterexamples in $\mathrm{Ac}(\mathcal{D}_\psi)$ to counterexamples in $\mathcal{D}$. Finally, counterexamples in $\mathrm{Ac}(\mathcal{D}_\psi)$ are computed by reducing the problem to a k shortest path problem, as in [HK07a]. Because $\mathrm{Ac}(\mathcal{D}_\psi)$ is acyclic, the complexity is lower than the corresponding problem in [HK07a].

It is worth mentioning that our technique can also be applied to pCTL formulas without nested path quantifiers.

Organization of the paper. Section 2 presents the necessary background on Markov chains (MC), Markov Decision Processes (MDP), and Linear Temporal Logic (LTL). Section 3 presents the definition of counterexamples and discusses the reduction from general LTL formulas to upper bounded probabilistic reachability properties, and the extraction of the maximizing MC in an MDP. Section 4 discusses desired properties of counterexamples. In Sections 5 and 6 we introduce the fundamentals on rails and torrents, the reduction of the original MC to the acyclic one, and our notion of significant diagnostic counterexamples. Section 7 then presents the techniques to actually compute counterexamples. In Section 8 we discuss related work and give final conclusions.

2 Preliminaries

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) constitute a formalism that combines nondeterministic and probabilistic choices. They are an important model in corporate finance, supply chain optimization, system verification and optimization. There are many slightly different variants of this formalism such as action-labeled MDPs [Bel57, FV97], probabilistic automata [SL95, SdV04]; we work with the state-labeled MDPs from [BdA95].

Definition 2.1. Let S be a finite set. A *probability distribution* on S is a function $p\colon S \to [0,1]$ such that $\sum_{s\in S} p(s) = 1$. We denote the set of all probability distributions on S by $\mathrm{Distr}(S)$. Additionally, we define the *Dirac distribution* on an element $s \in S$ as 1_s, i.e., $1_s(s) = 1$ and $1_s(t) = 0$ for all $t \in S \setminus \{s\}$.

Definition 2.2. A *Markov Decision Process* (MDP) is a quadruple $\mathcal{M} = (S, s_0, L, \tau)$, where

- S is the finite state space;
- $s_0 \in S$ is the initial state;
- L is a labeling function that associates to each state $s \in S$ a set $L(s)$ of propositional variables that are *valid* in s;
- $\tau\colon S \to \wp(\mathrm{Distr}(S))$ is a function that associates to each $s \in S$ a non-empty and finite subset of $\mathrm{Distr}(S)$ of probability distributions.

Definition 2.3. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP. We define a *successor* relation $\delta \subseteq S \times S$ by $\delta \triangleq \{(s,t) | \exists\, \pi \in \tau(s) \,.\, \pi(t) > 0\}$ and for each state $s \in S$ we define the sets

$$\mathrm{Paths}(\mathcal{M}, s) \triangleq \{t_0 t_1 t_2 \ldots \in S^\omega | t_0 = s \wedge \forall n \in \mathbb{N} \,.\, \delta(t_n, t_{n+1})\} \text{ and}$$

$$\mathrm{Paths}^\star(\mathcal{M}, s) \triangleq \{t_0 t_1 \ldots t_n \in S^\star | t_0 = s \wedge \forall\, 0 \leq i < n \,.\, \delta(t_n, t_{n+1})\}$$

of paths of $\mathcal{D}$ and finite paths of $\mathcal{D}$ respectively beginning at s. We usually omit $\mathcal{M}$ from the notation; we also abbreviate $\mathrm{Paths}(\mathcal{M}, s_0)$ as $\mathrm{Paths}(\mathcal{M})$ and $\mathrm{Paths}^\star(\mathcal{M}, s_0)$ as $\mathrm{Paths}^\star(\mathcal{M})$. For $\omega \in \mathrm{Paths}(s)$, we write the $(n+1)$-st state of ω as ω_n. As usual, we let $\mathcal{B}_s \subseteq \wp(\mathrm{Paths}(s))$ be the Borel σ-algebra on the cones

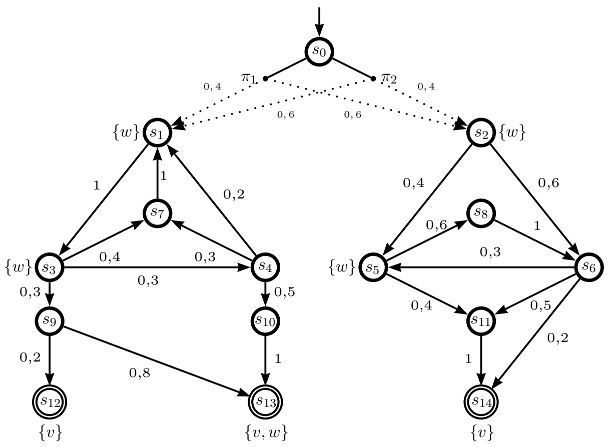


Fig. 3. Markov Decision Process

$\langle t_0 \dots t_n \rangle \triangleq \{\omega \in \text{Paths}(s) | \omega_0 = t_0 \wedge \dots \wedge \omega_n = t_n\}$. Additionally, for a set of finite paths $\Lambda \subseteq \text{Paths}^\star(s)$, we define $\langle \Lambda \rangle \triangleq \bigcup_{\sigma \in \Lambda} \langle \sigma \rangle$.

Figure 3 shows an MDP. Absorbing states (i.e., states s with $\tau(s) = \{1_s\}$) are represented by double lines. This MDP features a single nondeterministic decision, to be made in state s_0, namely π_1 and π_2.

Definition 2.4. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP, $s \in S$ and $\mathcal{A} \subseteq S$. We define the sets of paths and finite paths reaching $\mathcal{A}$ from s as

$$\text{Reach}(\mathcal{M}, s, \mathcal{A}) \triangleq \{\omega \in \text{Paths}(\mathcal{M}, s) \mid \exists_{i \geq 0}.\omega_i \in \mathcal{A}\} \text{ and}$$
$$\text{Reach}^\star(\mathcal{M}, s, \mathcal{A}) \triangleq \{\sigma \in \text{Paths}^\star(\mathcal{M}, s) \mid \text{last}(\sigma) \in \mathcal{A} \wedge \forall_{i \leq |\sigma|-1}.\sigma_i \notin \mathcal{A}\}$$

respectively. Note that $\text{Reach}^\star(\mathcal{M}, s, \mathcal{A})$ consists of those finite paths σ starting on s reaching $\mathcal{A}$ exactly once, at the end of the execution. It is easy to check that these sets are *prefix free*, i.e. contain finite paths such that none of them is a prefix of another one.

2.2 Schedulers

Schedulers (also called strategies, adversaries, or policies) resolve the nondeterministic choices in an MDP [PZ93, Var85, BdA95].

Definition 2.5. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP. A *scheduler* η on $\mathcal{M}$ is a function from $\text{Paths}^\star(\mathcal{M})$ to $\text{Distr}(\wp(\text{Distr}(S)))$ such that for all $\sigma \in \text{Paths}^\star(\mathcal{M})$ we have $\eta(\sigma) \in \text{Distr}(\tau(\text{last}(\sigma)))$. We denote the set of all schedulers on $\mathcal{M}$ by $\text{Sch}(\mathcal{M})$.

Note that our schedulers are randomized, i.e., in a finite path σ a scheduler chooses an element of $\tau(\text{last}(\sigma))$ probabilistically. Under a scheduler η, the probability that the next state reached after the path σ is t, equals $\sum_{\pi\in\tau(\text{last}(\sigma))}\eta(\sigma)(\pi)\cdot\pi(t)$. In this way, a scheduler induces a probability measure on $\mathcal{B}_s$ as usual.

Definition 2.6. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP and η a scheduler on $\mathcal{M}$. We define the probability measure $\mathbf{Pr}_\eta$ as the unique measure on $\mathcal{B}_{s_0}$ such that for all $s_0 s_1 \ldots s_n \in \text{Paths}^\star(\mathcal{M})$

$$\mathbf{Pr}_\eta(\langle s_0 s_1 \ldots s_n\rangle) = \prod_{i=0}^{n-1} \sum_{\pi\in\tau(s_i)} \eta(s_0 s_1 \ldots s_i)(\pi)\cdot\pi(s_{i+1}).$$

We now recall the notions of deterministic and memoryless schedulers.

Definition 2.7. Let $\mathcal{M}$ be an MDP and η a scheduler on $\mathcal{M}$. We say that η is *deterministic* if $\eta(\sigma)(\pi_i)$ is either 0 or 1 for all $\pi_i \in \tau(\text{last}(\sigma))$ and all $\sigma \in \text{Paths}^\star(\mathcal{M})$. We say that a scheduler is *memoryless* if for all finite paths σ_1, σ_2 of $\mathcal{M}$ with $\text{last}(\sigma_1) = \text{last}(\sigma_2)$ we have $\eta(\sigma_1) = \eta(\sigma_2)$

Definition 2.8. Let $\mathcal{M}$ be an MDP and $\Delta \in \mathcal{B}_{s_0}$. Then the *maximal probability* $\mathbf{Pr}^+$ and *minimal probability* $\mathbf{Pr}^-$ of Δ are defined by

$$\mathbf{Pr}^+(\Delta) \triangleq \sup_{\eta\in\text{Sch}(\mathcal{M})} \mathbf{Pr}_\eta(\Delta) \quad \text{and} \quad \mathbf{Pr}^-(\Delta) \triangleq \inf_{\eta\in\text{Sch}(\mathcal{M})} \mathbf{Pr}_\eta(\Delta).$$

A scheduler that attains $\mathbf{Pr}^+(\Delta)$ or $\mathbf{Pr}^-(\Delta)$ is called a *maximizing* or *minimizing* scheduler respectively.

2.3 Markov Chains

A (discrete time) *Markov chain* is an MDP associating exactly one probability distribution to each state. In this way nondeterministic choices are not longer allowed.

Definition 2.9 (Markov chain)**.** Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP. If $|\tau(s)| = 1$ for all $s \in S$, then we say that $\mathcal{M}$ is a *Markov chain* (MC).

In order to simplify notation we represent probabilistic transitions on MCs by means of a probabilistic matrix $\mathcal{P}$ instead of τ. Additionally, we denote by $\mathbf{Pr}_{\mathcal{D},s}$ the probability measure induced by a MC $\mathcal{D}$ with initial state s and we abbreviate $\mathbf{Pr}_{\mathcal{D},s_0}$ as $\mathbf{Pr}_{\mathcal{D}}$.

2.4 Linear Temporal Logic

Linear temporal logic (LTL) [MP91] is a modal temporal logic with modalities referring to time. In LTL is possible to encode formulas about the future of paths: a condition will eventually be true, a condition will be true until another fact becomes true, etc.

Definition 2.10. LTL is built up from the set of propositional variables $\mathcal{V}$, the logical connectives $\neg$, $\wedge$, and a temporal modal operator by the following grammar:

$$\phi ::= \mathcal{V} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathcal{U} \phi.$$

Using these operators we define $\vee, \rightarrow, \Diamond$, and $\Box$ in the standard way.

Definition 2.11. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP. We define satisfiability for paths ω in $\mathcal{M}$, propositional variables $v \in \mathcal{V}$, and LTL formulas ϕ, γ inductively by

$$\begin{array}{ll} \omega \models_{\mathcal{M}} v \quad \Leftrightarrow v \in L(\omega_0) & \omega \models_{\mathcal{M}} \phi \wedge \gamma \Leftrightarrow \omega \models_{\mathcal{M}} \phi \text{ and } \omega \models_{\mathcal{M}} \gamma \\ \omega \models_{\mathcal{M}} \neg\phi \Leftrightarrow \text{not}(\omega \models_{\mathcal{M}} \phi) & \omega \models_{\mathcal{M}} \phi \mathcal{U} \gamma \ \Leftrightarrow \exists_{i \geq 0}.\omega_{\downarrow i} \models_{\mathcal{M}} \gamma \text{ and } \forall_{0 \leq j < i}.\omega_{\downarrow j} \models_{\mathcal{M}} \phi \end{array}$$

where $\omega_{\downarrow i}$ is the i-th suffix of ω. When confusion is unlikely, we omit the subscript $\mathcal{M}$ on the satisfiability relation.

Definition 2.12. Let $\mathcal{M}$ be an MDP. We define the language $\text{Sat}_{\mathcal{M}}(\phi)$ associated to an LTL formula ϕ as the set of paths satisfying ϕ, i.e. $\text{Sat}_{\mathcal{M}}(\phi) \triangleq \{\omega \in \text{Paths}(\mathcal{M}) \mid \omega \models \phi\}$. Here we also generally omit the subscript $\mathcal{M}$.

We now define satisfiability of an LTL formula ϕ on an MDP $\mathcal{M}$. We say that $\mathcal{M}$ satisfies ϕ with probability at most p ($\mathcal{M} \models_{\leq p} \phi$) if the probability of getting an execution satisfying ϕ is at most p.

Definition 2.13. Let $\mathcal{M}$ be an MDP, ϕ an LTL formula and $p \in [0, 1]$. We define $\models_{\leq p}$ and $\models_{\geq p}$ by

$$\mathcal{M} \models_{\leq p} \phi \Leftrightarrow \mathbf{Pr}^{+}(\text{Sat}(\phi)) \leq p,$$
$$\mathcal{M} \models_{\geq p} \phi \Leftrightarrow \mathbf{Pr}^{-}(\text{Sat}(\phi)) \geq p.$$

We define $\mathcal{M} \models_{< p} \phi$ and $\mathcal{M} \models_{> p} \phi$ in a similar way. In case the MDP is fully probabilistic, i.e., an MC, the satisfiability problem is reduced to $\mathcal{M} \models_{\bowtie p} \phi \Leftrightarrow \mathbf{Pr}_{\mathcal{M}}(\text{Sat}(\phi)) \bowtie p$, where $\bowtie \in \{<, \leq, >, \geq\}$.

3 Counterexamples

In this section, we define what counterexamples are and how the problem of finding counterexamples to a general LTL property over Markov Decision Processes reduces to finding counterexamples to reachability problems over Markov chains.

Definition 3.1 (Counterexamples)**.** Let $\mathcal{M}$ be an MDP and ϕ an LTL formula. A *counterexample* to $\mathcal{M} \models_{\leq p} \phi$ is a measurable set $\mathcal{C} \subseteq \text{Sat}(\phi)$ such that $\mathbf{Pr}^{+}(\mathcal{C}) > p$. Counterexamples to $\mathcal{M} \models_{< p} \phi$ are defined similarly.

Counterexamples to $\mathcal{M} \models_{>p} \phi$ and $\mathcal{M} \models_{\geq p} \phi$ cannot be defined straightforwardly as it is always possible to find a set $\mathcal{C} \subseteq \mathrm{Sat}(\phi)$ such that $\mathbf{Pr}^{-}(\mathcal{C}) \leq p$ or $\mathbf{Pr}^{-}(\mathcal{C}) < p$, note that the empty set trivially satisfies it. Therefore, the best way to find counterexamples to lower bounded probabilities is to find counterexamples to the dual properties $\mathcal{M} \models_{<1-p} \neg\phi$ and $\mathcal{M} \models_{\leq 1-p} \neg\phi$. That is, while for upper bounded probabilities, a counterexample is a set of paths satisfying the property with mass probability beyond the bound, for lower bounded probabilities the counterexample is a set of paths that *does not* satisfy the property with sufficient probability.

Example 1. Consider the MDP $\mathcal{M}$ of Figure 4 and the LTL formula $\Diamond v$. It is easy to check that $\mathcal{M} \not\models_{<1} \Diamond v$. The set $\mathcal{C} = \mathrm{Sat}(\Diamond v) = \{\rho \in \mathrm{Paths}(s_0) | \exists_{i \geq 0} . \rho = s_0(s_1)^i(s_4)^\omega\} \cup \{\rho \in \mathrm{Paths}(s_0) | \exists_{i \geq 0} . \rho = s_0(s_3)^i(s_5)^\omega\}$ is a counterexample. Note that $\mathbf{Pr}_\eta(\mathcal{C}) = 1$ where η is any deterministic scheduler on $\mathcal{M}$ satisfying $\eta(s_0) = \pi_1$.

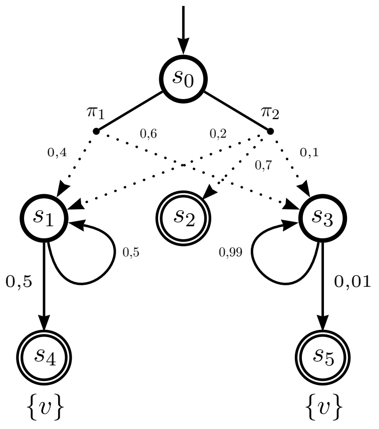


Fig. 4

LTL formulas are actually checked by reducing the model checking problem to a reachability problem [Alf97]. For checking upper bounded probabilities, the LTL formula is translated into an equivalent deterministic Rabin automaton and composed with the MDP under verification. On the obtained MDP, the set of states forming accepting end components (SCC that traps accepting conditions with probability 1) are identified. The maximum probability of the LTL property on the original MDP is the same as the maximum probability of reaching a state of an accepting end component in the final MDP. Hence, from now on we will focus on counterexamples to properties of the form $\mathcal{M} \models_{\leq p} \Diamond\psi$ or $\mathcal{M} \models_{<p} \Diamond\psi$, where ψ is a propositional formula, i.e., a formula without temporal operators.

In the following, it will be useful to identify the set of states in which a propositional property is valid.

Definition 3.2. Let $\mathcal{M}$ be an MDP. We define the state language $\mathrm{Sat}_{\mathcal{M}}(\psi)$ associated to a propositional formula ψ as the set of states satisfying ψ, i.e., $\mathrm{Sat}_{\mathcal{M}}(\psi) \triangleq \{s \in S \mid s \models \psi\}$, where $\models$ has the obvious satisfaction meaning for states. As usual, we generally omit the subscript $\mathcal{M}$.

We will show now that, in order to find a counterexample to a property in an MDP with respect to an upper bound, it suffices to find a counterexample for the MC *induced* by the maximizing scheduler. The maximizing scheduler turns out to be deterministic and memoryless [BdA95]; consequently the induced Markov chain can be easily extracted from the MDP as follows.

Definition 3.3. Let $\mathcal{M} = (S, s_0, \tau, L)$ be an MDP and η a deterministic memoryless scheduler. Then we define the MC induced by η as $\mathcal{M}_\eta = (S, s_0, \mathcal{P}_\eta, L)$ where $\mathcal{P}_\eta(s,t) = (\eta(s))(t)$ for all $s, t \in S$.

Now we state that finding counterexamples to upper bounded probabilistic reachability LTL properties on MDPs can be reduced to finding counterexamples to upper bounded probabilistic reachability LTL properties on MCs.

Theorem 3.4. *Let $\mathcal{M}$ be an* MDP*, ψ a propositional formula and $p \in [0,1]$. Then, there is a maximizing (deterministic memoryless) scheduler η such that $\mathcal{M} \models_{\leq p} \Diamond\psi \Leftrightarrow \mathcal{M}_\eta \models_{\leq p} \Diamond\psi$. Moreover, if $\mathcal{C}$ is a counterexample to $\mathcal{M}_\eta \models_{\leq p} \Diamond\psi$ then $\mathcal{C}$ is also a counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$.*

Note that η can be computed by solving a linear minimization problem [BdA95]. See Section 7.1.

4 Representative Counterexamples, Partitions and Witnesses

The notion of counterexample from Definition 3.1 is very broad: just an arbitrary (measurable) set of paths with high enough mass probability. To be useful as a debugging tool (and in fact to be able to present the counterexample to a user), we need counterexamples with specific properties. We will partition counterexamples (or rather, representative counterexamples) in witnesses and list five informal properties that we consider valuable in order to increase the quality of witnesses as a debugging tool.

We first note that for reachability properties it is sufficient to consider counterexamples that consist of finite paths.

Definition 4.1 (Representative counterexamples)**.** Let $\mathcal{M}$ be an MDP, ψ a propositional formula and $p \in [0,1]$. A *representative counterexample* to $\mathcal{M} \models_{\leq p} \Diamond\psi$ is a set $\mathcal{C} \subseteq \mathrm{Reach}^\star(\mathcal{M}, \mathrm{Sat}(\psi))$ such that $\mathbf{Pr}^+(\langle\mathcal{C}\rangle) > p$. We denote the set of all representative counterexamples to $\mathcal{M} \models_{\leq p} \Diamond\psi$ by $\mathcal{R}(\mathcal{M}, p, \psi)$.

Theorem 4.2. *Let $\mathcal{M}$ be an* MDP*, ψ a propositional formula and $p \in [0,1]$. If $\mathcal{C}$ is a representative counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$, then $\langle\mathcal{C}\rangle$ is a counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$. Furthermore, there exists a counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$ if and only if there exists a representative counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$.*

Following [HK07a], we present the notions of *minimum counterexample*, *strongest evidence* and *most indicative counterexamples.*

Definition 4.3 (Minimum counterexample)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0,1]$. We say that $\mathcal{C} \in \mathcal{R}(\mathcal{D}, p, \psi)$ is a *minimum counterexample* if $|\mathcal{C}| \leq |\mathcal{C}'|$, for all $\mathcal{C}' \in \mathcal{R}(\mathcal{D}, p, \psi)$.

Definition 4.4 (Strongest evidence)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0,1]$. A *strongest evidence* to $\mathcal{D} \not\models_{\leq p} \Diamond\psi$ is a finite path $\sigma \in \mathrm{Reach}^\star(\mathcal{D}, \mathrm{Sat}(\psi))$ such that $\mathbf{Pr}_{\mathcal{D}}(\langle\sigma\rangle) \geq \mathbf{Pr}_{\mathcal{D}}(\langle\rho\rangle)$, for all $\rho \in \mathrm{Reach}^\star(\mathcal{D}, \mathrm{Sat}(\psi))$.

Definition 4.5 (Most indicative counterexample)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0,1]$. We call $\mathcal{C} \in \mathcal{R}(\mathcal{D}, p, \psi)$ a *most indicative counterexample* if it is minimum and $\mathbf{Pr}_{\mathcal{D}}(\langle \mathcal{C} \rangle) \geq \mathbf{Pr}_{\mathcal{D}}(\langle \mathcal{C}' \rangle)$, for all minimum counterexamples $\mathcal{C}' \in \mathcal{R}(\mathcal{D}, p, \psi)$.

Unfortunately, very often most indicative counterexamples are very large (even infinite), many of its elements have insignificant measure and elements can be extremely similar to each other (consequently providing the same diagnostic information). Even worse, sometimes the finite paths with highest probability do not exhibit the way in which the system accumulates higher probability to reach the undesired property (and consequently where an error occurs with higher probability). For these reasons, we are of the opinion that representative counterexamples are still too general in order to be useful as feedback information. We approach this problem by refining a representative counterexample into sets of finite paths following a "similarity" criteria (introduced in Section 5). These sets are called *witnesses* of the counterexample.

Recall that a set Y of nonempty sets is a partition of X if the elements of Y cover X and are pairwise disjoint. We define counterexample partitions in the following way.

Definition 4.6 (Counterexample partitions and witnesses)**.** Let $\mathcal{M}$ be an MDP, ψ a propositional formula, $p \in [0,1]$, and $\mathcal{C}$ a representative counterexample to $\mathcal{M} \models_{\leq p} \Diamond\psi$. A *counterexample partition* $W_{\mathcal{C}}$ is a partition of $\mathcal{C}$. We call the elements of $W_{\mathcal{C}}$ *witnesses.*

Since not every partition generates useful witnesses (from the debugging perspective), we now state five informal properties that we consider valuable in order to improve the diagnostic information provided by witnesses. In Section 7 we show how to partition the representative counterexample in order to obtain witnesses satisfying most of these properties.

Similarity: Elements of a witness should provide similar debugging information.
Accuracy: Witnesses with higher probability should exhibit evolutions of the system with higher probability of containing errors.
Originality: Different witnesses should provide different debugging information.
Significance: Witnesses should be as closed to the counterexample as possible (their mass probability should be as closed as possible to the bound p).
Finiteness: The number of witnesses of a counterexample partition should be finite.

5 Rails and Torrents

As argued before we consider that representative counterexamples are excessively general to be useful as feedback information. Therefore, we group finite paths of a representative counterexample in witnesses if they are "similar enough". We will consider finite paths that behave the same outside SCCs of the system as providing similar feedback information.

In order to formalize this idea, we first reduce the original MC $\mathcal{D}$ to an acyclic MC preserving reachability probabilities. We do so by removing all SCCs K of $\mathcal{D}$ keeping just *input states* of K. In this way, we get a new acyclic MC denoted by $\mathrm{Ac}(\mathcal{D})$. The probability matrix of the Markov chain relates input states of each SCC to its *output states* with the reachability probability between these states in $\mathcal{D}$. Secondly, we establish a map between finite paths σ in $\mathrm{Ac}(\mathcal{D})$ (*rails*) and sets of paths W_σ in $\mathcal{D}$ (*torrents*). Each torrent contains finite paths that are similar, i.e., behave the same outside SCCs. We conclude the section showing that the probability of σ is equal to the mass probability of W_σ.

Reduction to Acyclic Markov Chains

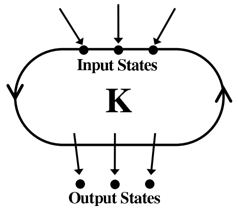
Consider an MC $\mathcal{D} = (S, s_0, \mathcal{P}, L)$. Recall that a subset $\mathrm{K} \subseteq S$ is called *strongly connected* if for every $s, t \in \mathrm{K}$ there is a finite path from s to t. Additionally K is called a *strongly connected component* (SCC) if it is a maximally (with respect to $\subseteq$) strongly connected subset of S.

Note that every state is a member of exactly one SCC of $\mathcal{D}$ (even those states that are not involved in cycles, since the trivial finite path s connects s to itself). From now on we let $\mathrm{SCC}^\star$ be the set of non trivial strongly connected components of an MC, i.e., those composed of more than one state.

A Markov chain is called *acyclic* if it contains only trivial SCCs. Note that an acyclic Markov chain still has absorbing states.

Definition 5.1 (Input and Output states)**.** Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC. Then, for each $\mathrm{SCC}^\star$ K of $\mathcal{D}$, we define the sets $\mathrm{Inp}_\mathrm{K} \subseteq S$ of all states in K that have an incoming transition from a state outside of K and $\mathrm{Out}_\mathrm{K} \subseteq S$ of all states outside of K that have an incoming transition from a state of K in the following way

$$\begin{aligned} \mathrm{Inp}_\mathrm{K} &\triangleq \{t \in \mathrm{K} \mid \exists\, s \in S \setminus \mathrm{K} \,.\, \mathcal{P}(s,t) > 0\}, \\ \mathrm{Out}_\mathrm{K} &\triangleq \{s \in S \setminus \mathrm{K} \mid \exists\, t \in \mathrm{K} \,.\, \mathcal{P}(t,s) > 0\}. \end{aligned}$$



We also define for each $\mathrm{SCC}^\star$ K an MC related to K as $\mathcal{D}_\mathrm{K} \triangleq (\mathrm{K} \cup \mathrm{Out}_\mathrm{K}, s_\mathrm{K}, \mathcal{P}_\mathrm{K}, L_\mathrm{K})$ where s_K is any state in Inp_K, $L_\mathrm{K}(s) \triangleq L(s)$, and $\mathcal{P}_\mathrm{K}(s,t)$ is equal to $\mathcal{P}(s,t)$ if $s \in \mathrm{K}$ and equal to 1_s otherwise. Additionally, for every state s involved in non trivial SCCs we define SCC_s^+ as $\mathcal{D}_\mathrm{K}$, where K is the $\mathrm{SCC}^\star$ of $\mathcal{D}$ such that $s \in \mathrm{K}$.

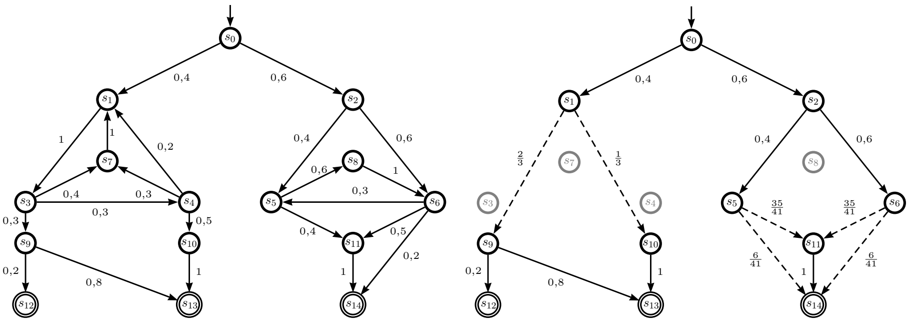
Now we are able to define an acyclic MC $\mathrm{Ac}(\mathcal{D})$ related to $\mathcal{D}$.

Definition 5.2. Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be a MC. We define $\text{Ac}(\mathcal{D}) \triangleq (S', s_0, \mathcal{P}', L')$ where

- $S' \triangleq \overbrace{S \setminus \bigcup_{\text{K} \in \text{SCC}^\star} \text{K}}^{S_{\text{com}}} \cup \overbrace{\bigcup_{\text{K} \in \text{SCC}^\star} \text{Inp}_{\text{K}}}^{S_{\text{inp}}}$,
- $L' \triangleq L_{|_{S'}}$,
- $\mathcal{P}'(s,t) \triangleq \begin{cases} \mathcal{P}(s,t) & \text{if } s \in S_{com}, \\ \mathbf{Pr}_{\mathcal{D},s}(\text{Reach}(\text{SCC}_s^+, s, \{t\})) & \text{if } s \in S_{inp} \wedge t \in \text{Out}_{\text{SCC}_s^+}, \\ 1 & \text{if } s \in S_{inp} \wedge \emptyset = \text{Out}_{\text{SCC}_s^+} \wedge t = s, \\ 0 & \text{otherwise.} \end{cases}$

Note that $\text{Ac}(\mathcal{D})$ is indeed acyclic.

Example 2. Consider the MC $\mathcal{D}$ of Figure 5(a). The strongly connected components of $\mathcal{D}$ are $\text{K}_1 \triangleq \{s_1, s_3, s_4, s_7\}$, $\text{K}_2 \triangleq \{s_5, s_6, s_8\}$ and the singletons $\{s_0\}$, $\{s_2\}$, $\{s_9\}$, $\{s_{10}\}$, $\{s_{11}\}$, $\{s_{12}\}$, $\{s_{13}\}$, and $\{s_{14}\}$. The input states of K_1 are $\text{Inp}_{\text{K}_1} = \{s_1\}$ and its output states are $\text{Out}_{\text{K}_1} = \{s_9, s_{10}\}$. For K_2, $\text{Inp}_{\text{K}_2} = \{s_5, s_6\}$ and $\text{Out}_{\text{K}_2} = \{s_{11}, s_{14}\}$. The reduced acyclic MC of $\mathcal{D}$ is shown in Figure 5(b).



(a) Original MC (b) Derived Acyclic MC

Fig. 5

Rails and Torrents

We now relate (finite) paths in $\text{Ac}(\mathcal{D})$ (rails) to sets of paths in $\mathcal{D}$ (torrents).

Definition 5.3 (Rails)**.** Let $\mathcal{D}$ be an MC. A finite path $\sigma \in \text{Paths}^\star(\text{Ac}(\mathcal{D}))$ will be called a *rail of* $\mathcal{D}$.

Consider a rail σ, i.e., a finite path of $\text{Ac}(\mathcal{D})$. We will use σ to represent those paths ω of $\mathcal{D}$ that behave "similar to" σ outside SCCs of $\mathcal{D}$. Naively, this means that σ is a subsequence of ω. There are two technical subtleties to deal with: every input state in σ must be the first state in its SCC in ω (freshness) and every SCC visited by ω must be also visited by σ (inertia) (see Definition 5.5). We need these extra conditions to make sure that no path ω behaves "similar to" two distinct rails (see Lemma 5.7).

Recall that given a finite sequence σ and a (possible infinite) sequence ω, we say that σ is a *subsequence* of ω, denoted by $\sigma \sqsubseteq \omega$, if and only if there exists a strictly increasing function $f : \{0, 1, \ldots, |\sigma| - 1\} \to \{0, 1, \ldots, |\omega| - 1\}$ such that $\forall_{0 \leq i < |\sigma|} . \sigma_i = \omega_{f(i)}$. If ω is an infinite sequence, we interpret the codomain of f as $\mathbb{N}$. In case f is such a function we write $\sigma \sqsubseteq_f \omega$.

Definition 5.4. Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC. On S we consider the equivalence relation $\underset{\mathcal{D}}{\sim}$ satisfying $s \underset{\mathcal{D}}{\sim} t$ if and only if s and t are in the same strongly connected component. Again, we usually omit the subscript $\mathcal{D}$ from the notation.

The following definition refines the notion of subsequence, taking care of the two technical subtleties noted above.

Definition 5.5. Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC, ω a (finite) path of $\mathcal{D}$, and $\sigma \in \mathrm{Paths}^{\star}(\mathrm{Ac}(\mathcal{D}))$ a finite path of $\mathrm{Ac}(\mathcal{D})$. Then we write $\sigma \preceq \omega$ if there exists $f : \{0, 1, \ldots, |\sigma| - 1\} \to \mathbb{N}$ such that $\sigma \sqsubseteq_f \omega$ and

$$\forall_{0 \leq j < f(i)} : \omega_{f(i)} \not\sim \omega_j ;\ \text{for all } i = 0, 1, \ldots |\sigma| - 1, \quad [\textit{Freshness property}]$$
$$\forall_{f(i) < j < f(i+1)} : \omega_{f(i)} \sim \omega_j ;\ \text{for all } i = 0, 1, \ldots |\sigma| - 2. \quad [\textit{Inertia property}]$$

In case f is such a function we write $\sigma \preceq_f \omega$.

Example 3. Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be the MC of Figure 5(a) and take $\sigma = s_0 s_2 s_6 s_{14}$. Then for all $i \in \mathbb{N}$ we have $\sigma \preceq_{f_i} \omega_i$ where $\omega_i = s_0 s_2 s_6 (s_5 s_8 s_6)^i s_{14}$ and $f_i(0) \triangleq 0$, $f_i(1) \triangleq 1$, $f_i(2) \triangleq 2$, and $f_i(3) \triangleq 3 + 3i$. Additionally, $\sigma \not\preceq s_0 s_2 s_5 s_8 s_6 s_{14}$ since for all f satisfying $\sigma \sqsubseteq_f s_0 s_2 s_5 s_8 s_6 s_{14}$ we must have $f(2) = 5$; this implies that f does not satisfy the freshness property. Finally, note that $\sigma \not\preceq s_0 s_2 s_6 s_{11} s_{14}$ since for all f satisfying $\sigma \sqsubseteq_f s_0 s_2 s_6 s_{11} s_{14}$ we must have $f(2) = 2$; this implies that f does not satisfy the inertia property.

We now give the formal definition of torrents.

Definition 5.6 (Torrents)**.** Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC and σ a sequence of states in S. We define the function Torr by

$$\mathrm{Torr}(\mathcal{D}, \sigma) \triangleq \{\omega \in \mathrm{Paths}(\mathcal{D}) \mid \sigma \preceq \omega\}.$$

We call $\mathrm{Torr}(\mathcal{D}, \sigma)$ the *torrent* associated to σ.

We now show that torrents are disjoint (Lemma 5.7) and that the probability of a rail is equal to the probability of its associated torrent (Theorem 5.10). For this last result, we first show that torrents can be represented as the disjoint union of cones of finite paths. We call these finite paths *generators* of the torrent (Definition 5.8).

Lemma 5.7. *Let $\mathcal{D}$ be an* MC. *For every $\sigma, \rho \in \mathrm{Paths}^{\star}(\mathrm{Ac}(\mathcal{D}))$ we have*

$$\sigma \neq \rho \Rightarrow \mathrm{Torr}(\mathcal{D}, \sigma) \cap \mathrm{Torr}(\mathcal{D}, \rho) = \emptyset$$

Definition 5.8 (Torrent Generators)**.** Let $\mathcal{D}$ be an MC. Then we define for every rail $\sigma \in \text{Paths}^\star(\text{Ac}(\mathcal{D}))$ the set

$$\text{TorrGen}(\mathcal{D}, \sigma) \triangleq \{\rho \in \text{Paths}^\star(\mathcal{D}) \mid \exists f : \sigma \preceq_f \rho \wedge f(|\sigma| - 1) = |\rho| - 1\}.$$

In the example from the Introduction (see Figure 1), $s_0 s_1 s_3$ and $s_0 s_2 s_4$ are rails. Their associated torrents are, respectively, $\{s_0 s_1^n s_3^\omega \mid n \in \mathbb{N}^*\}$ and $\{s_0 s_2^n s_4^\omega \mid n \in \mathbb{N}^*\}$ (note that s_3 and s_4 are absorbing states), i.e. the paths going left and the paths going right. The generators of the first torrent are $\{s_0 s_1^n s_3 \mid n \in \mathbb{N}^*\}$ and similarly for the second torrent.

Lemma 5.9. *Let $\mathcal{D}$ be an* MC *and $\sigma \in \text{Paths}^\star(\text{Ac}(\mathcal{D}))$ a rail of $\mathcal{D}$. Then we have*

$$\text{Torr}(\mathcal{D}, \sigma) = \biguplus_{\rho \in \text{TorrGen}(\mathcal{D}, \sigma)} \langle \rho \rangle.$$

Theorem 5.10. *Let $\mathcal{D}$ be an* MC*. Then for every rail $\sigma \in \text{Paths}^\star(\text{Ac}(\mathcal{D}))$ we have*

$$\mathbf{Pr}_{\text{Ac}(\mathcal{D})}(\langle \sigma \rangle) = \mathbf{Pr}_{\mathcal{D}}(\text{Torr}(\mathcal{D}, \sigma)).$$

6 Significant Diagnostic Counterexamples

So far we have formalized the notion of paths behaving similarly (i.e., behaving the same outside SCCs) in an MC $\mathcal{D}$ by removing all SCC of $\mathcal{D}$, obtaining $\text{Ac}(\mathcal{D})$. A representative counterexample to $\text{Ac}(\mathcal{D}) \models_{\leq p} \Diamond\psi$ gives rise to a representative counterexample to $\mathcal{D} \models_{\leq p} \Diamond\psi$ in the following way: for every finite path σ in the representative counterexample to $\text{Ac}(\mathcal{D}) \models_{\leq p} \Diamond\psi$ the set $\text{TorrGen}(\mathcal{D}, \sigma)$ is a witness, then we obtain the desired representative counterexample to $\mathcal{D} \models_{\leq p} \Diamond\psi$ by taking the union of these witnesses.

Before giving a formal definition, there is still one technical issue to resolve: we need to be sure that by removing SCCs we are not discarding useful information. Because torrents are built from rails, we need to make sure that when we discard SCCs, we do not discard rails that reach ψ.

We achieve this by first making states satisfying ψ absorbing. Additionally, we make absorbing states from which it is not possible to reach ψ. Note that this does not affect counterexamples.

Definition 6.1. Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC and ψ a propositional formula. We define the MC $\mathcal{D}_\psi \triangleq (S, s_0, \mathcal{P}_\psi, L)$, with

$$\mathcal{P}_\psi(s, t) \triangleq \begin{cases} 1 & \text{if } s \notin \text{Sat}_\Diamond(\psi) \wedge s = t, \\ 1 & \text{if } s \in \text{Sat}(\psi) \wedge s = t, \\ \mathcal{P}(s,t) & \text{if } s \in \text{Sat}_\Diamond(\psi) - \text{Sat}(\psi), \\ 0 & \text{otherwise}, \end{cases}$$

where $\text{Sat}_\Diamond(\psi) \triangleq \{s \in S \mid \mathbf{Pr}_{\mathcal{D},s}(\text{Reach}(\mathcal{D}, s, \text{Sat}(\psi))) > 0\}$ is the set of states reaching ψ in $\mathcal{D}$.

The following theorem shows the relation between paths, finite paths, and probabilities of $\mathcal{D}$, $\mathcal{D}_\psi$, and $\mathrm{Ac}(\mathcal{D}_\psi)$. Most importantly, the probability of a rail σ (in $\mathrm{Ac}(\mathcal{D}_\psi)$) is equal to the probability of its associated torrent (in $\mathcal{D}$) (item 5 below) and the probability of $\Diamond\psi$ is not affected by reducing $\mathcal{D}$ to $\mathrm{Ac}(\mathcal{D}_\psi)$ (item 6 below).

Note that a rail σ is always a finite path in $\mathrm{Ac}(\mathcal{D}_\psi)$, but that we can talk about its associated torrent $\mathrm{Torr}(\mathcal{D}_\psi, \sigma)$ in $\mathcal{D}_\psi$ and about its associated torrent $\mathrm{Torr}(\mathcal{D}, \sigma)$ in $\mathcal{D}$. The former exists for technical convenience; it is the latter that we are ultimately interested in. The following theorem also shows that for our purposes, viz. the definition of the generators of the torrent and the probability of the torrent, there is no difference (items 3 and 4 below).

Theorem 6.2. *Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an* MC *and ψ a propositional formula. Then for every $\sigma \in \mathrm{Paths}^\star(\mathcal{D}_\psi)$*

1. $\mathrm{Reach}^\star(\mathcal{D}_\psi, s_0, \mathrm{Sat}(\psi)) = \mathrm{Reach}^\star(\mathcal{D}, s_0, \mathrm{Sat}(\psi))$,
2. $\mathbf{Pr}_{\mathcal{D}_\psi}(\langle\sigma\rangle) = \mathbf{Pr}_{\mathcal{D}}(\langle\sigma\rangle)$,
3. $\mathrm{TorrGen}(\mathcal{D}_\psi, \sigma) = \mathrm{TorrGen}(\mathcal{D}, \sigma)$,
4. $\mathbf{Pr}_{\mathcal{D}_\psi}(\mathrm{Torr}(\mathcal{D}_\psi, \sigma)) = \mathbf{Pr}_{\mathcal{D}}(\mathrm{Torr}(\mathcal{D}, \sigma))$,
5. $\mathbf{Pr}_{\mathrm{Ac}(\mathcal{D}_\psi)}(\langle\sigma\rangle) = \mathbf{Pr}_{\mathcal{D}}(\mathrm{Torr}(\mathcal{D}, \sigma))$,
6. $\mathrm{Ac}(\mathcal{D}_\psi) \models_{\leq p} \Diamond\psi$ *if and only if* $\mathcal{D} \models_{\leq p} \Diamond\psi$*, for any* $p \in [0, 1]$.

Proof. Straightforward □

Definition 6.3 (Torrent-Counterexamples)**.** Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an MC, ψ a propositional formula, and $p \in [0, 1]$ such that $\mathcal{D} \not\models_{\leq p} \Diamond\psi$. Let $\mathcal{C}$ be a representative counterexample to $\mathrm{Ac}(\mathcal{D}_\psi) \models_{\leq p} \Diamond\psi$. We define the set

$$\mathrm{TorRepCount}(\mathcal{C}) \triangleq \{\mathrm{TorrGen}(\mathcal{D}, \sigma) \mid \sigma \in \mathcal{C}\}.$$

We call the set $\mathrm{TorRepCount}(\mathcal{C})$ a *torrent-counterexample* of $\mathcal{C}$. Note that this set is a partition of a representative counterexample to $\mathcal{D} \models_{\leq p} \Diamond\psi$. Additionally, we denote by $\mathcal{R}_t(\mathcal{D}, p, \psi)$ to the set of all torrent-counterexamples to $\mathcal{D} \models_{\leq p} \Diamond\psi$, i.e., $\{\mathrm{TorRepCount}(\mathcal{C}) \mid \mathcal{C} \in \mathcal{R}(\mathrm{Ac}(\mathcal{D}), p, \psi)\}$.

Theorem 6.4. *Let $\mathcal{D} = (S, s_0, \mathcal{P}, L)$ be an* MC*, ψ a propositional formula, and $p \in [0, 1]$ such that $\mathcal{D} \not\models_{\leq p} \Diamond\psi$. Take $\mathcal{C}$ a representative counterexample to $\mathrm{Ac}(\mathcal{D}_\psi) \models_{\leq p} \Diamond\psi$. Then the set of finite paths $\biguplus_{W \in \mathrm{TorRepCount}(\mathcal{C})} W$ is a representative counterexample to $\mathcal{D} \models_{\leq p} \Diamond\psi$.*

Note that for each $\sigma \in \mathcal{C}$ we get a witness $\mathrm{TorrGen}(\mathcal{D}, \sigma)$. Also note that the number of rails is finite, so there are also only finitely many witnesses.

Following [HK07a], we extend the notions of *minimum counterexamples* and *strongest evidence.*

Definition 6.5 (Minimum torrent-counterexample)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0, 1]$. We say that $\mathcal{C}_t \in \mathcal{R}_t(\mathcal{D}, p, \psi)$ is a *minimum torrent-counterexample* if $|\mathcal{C}_t| \leq |\mathcal{C}'_t|$, for all $\mathcal{C}'_t \in \mathcal{R}_t(\mathcal{D}, p, \psi)$.

Definition 6.6 (Strongest torrent-evidence)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0,1]$. A *strongest torrent-evidence* to $\mathcal{D} \not\models_{\leq p} \Diamond\psi$ is a torrent $\mathrm{Torr}(\mathcal{D},\sigma)$ such that $\sigma \in \mathrm{Paths}^\star(\mathrm{Ac}(\mathcal{D}_\psi))$ and $\mathbf{Pr}_\mathcal{D}(\mathrm{Torr}(\mathcal{D},\sigma)) \geq \mathbf{Pr}_\mathcal{D}(\mathrm{Torr}(\mathcal{D},\rho))$ for all $\rho \in \mathrm{Paths}^\star(\mathrm{Ac}(\mathcal{D}_\psi))$.

Now we define our notion of significant diagnostic counterexamples. It is the generalization of most indicative counterexample from [HK07a] to our setting.

Definition 6.7 (Most indicative torrent-counterexample)**.** Let $\mathcal{D}$ be an MC, ψ a propositional formula and $p \in [0,1]$. We call $\mathcal{C}_t \in \mathcal{R}_t(\mathcal{D},p,\psi)$ a *most indicative torrent-counterexample* if it is a minimum torrent-counterexample and $\mathbf{Pr}(\bigcup_{T\in\mathcal{C}_t}\langle T\rangle) \geq \mathbf{Pr}(\bigcup_{T\in\mathcal{C}'_t}\langle T\rangle)$ for all minimum torrent-counterexamples $\mathcal{C}'_t \in \mathcal{R}_t(\mathcal{D},p,\psi)$.

Note that in our setting, as in [HK07a], a minimal torrent-counterexample $\mathcal{C}$ consists of the $|\mathcal{C}|$ strongest torrent-evidences.

By Theorem 6.4 it is possible to obtain strongest torrent-evidence and most indicative torrent-counterexamples of an MC $\mathcal{D}$ by obtaining strongest evidence and most indicative counterexamples of $\mathrm{Ac}(\mathcal{D}_\psi)$ respectively.

7 Computing Counterexamples

In this section we show how to compute most indicative torrent-counterexamples. We also discuss what information to present to the user: how to present witnesses and how to deal with overly large strongly connected components.

7.1 Maximizing Schedulers

The calculation of the maximal probability on a reachability problem can be performed by solving a linear minimization problem [BdA95, dA97]. This minimization problem is defined on a system of inequalities that has a variable x_i for each different state s_i and an inequality $\sum_j \pi(s_j)\cdot x_j \leq x_i$ for each distribution $\pi \in \tau(s_i)$. The maximizing (deterministic memoryless) scheduler η can be easily extracted out of such system of inequalities after obtaining the solution. If $p_0,\ldots,p_n$ are the values that minimize $\sum_i x_i$ in the previous system, then η is such that, for all s_i, $\eta(s_i) = \pi$ whenever $\sum_j \pi(s_j)\cdot p_j = p_i$. In the following we denote $\mathbf{P}_{s_i}[\Diamond\psi] \triangleq x_i$.

7.2 Computing Most Indicative Torrent-Counterexamples

We divide the computation of most indicative torrent-counterexamples to $\mathcal{M} \models_{\leq p} \Diamond\psi$ in three stages: *pre-processing*, SCC *analysis*, and *searching*.

Pre-processing stage. We first modify the original MC $\mathcal{D}$ by making all states in $\mathrm{Sat}(\psi) \cup S \setminus \mathrm{Sat}_\Diamond(\psi)$ absorbing. In this way we obtain the MC $\mathcal{D}_\psi$ from Definition 6.1. Note that we do not have to spend additional computational resources to compute this set, since $\mathrm{Sat}_\Diamond(\psi) = \{s \in S \mid \mathbf{P}_s[\Diamond\psi] > 0\}$ and hence all required data is already available from the LTL model checking phase.

SCC **analysis stage.** We remove all SCCs K of $\mathcal{D}_\psi$ keeping just *input states* of K, getting the acyclic MC $\mathrm{Ac}(\mathcal{D}_\psi)$ according to Definition 5.2.

To compute this, we first need to find the SCCs of $\mathcal{D}_\psi$. There exists several well known algorithms to achieve this: Kosaraju's, Tarjan's, Gabow's algorithms (among others). We also have to compute the reachability probability from input states to output states of every SCC. This can be done by using steady state analysis techniques [Cas93].

Searching stage. To find most indicative torrent-counterexamples in $\mathcal{D}$, we find most indicative counterexamples in $\mathrm{Ac}(\mathcal{D}_\psi)$. For this we use the same approach as [HK07a], turning the MC into a weighted digraph to replace the problem of finding the finite path with highest probability by a shortest path problem. The nodes of the digraph are the states of the MC and there is an edge between s and t if $\mathcal{P}(s,t) > 0$. The weight of such an edge is $-\log(\mathcal{P}(s,t))$.

Finding the most indicative counterexample in $\mathrm{Ac}(\mathcal{D}_\psi)$ is now reduced to finding k shortest paths. As explained in [HK07a], our algorithm has to compute k on the fly. Eppstein's algorithm [Epp98] produces the k shortest paths in general in $O(m+n\log n+k)$, where m is the number of nodes and n the number of edges. In our case, since $\mathrm{Ac}(\mathcal{D}_\psi)$ is acyclic, the complexity decreases to $O(m+k)$.

7.3 Debugging Issues

Representative finite paths. What we have computed so far is a most indicative counterexample to $\mathrm{Ac}(\mathcal{D}_\psi) \models_{\leq p} \Diamond\psi$. This is a finite set of rails, i.e., a finite set of paths in $\mathrm{Ac}(\mathcal{D}_\psi)$. Each of these paths σ represents a witness $\mathrm{TorrGen}(\mathcal{D},\sigma)$. Note that this witness itself has usually infinitely many elements.

In practice, one has to display a witness to the user. The obvious way would be to show the user the rail σ. This, however, may be confusing to the user as σ is not a finite path of the original Markov Decision Process. Instead of presenting the user with σ, we therefore show the user the finite path of $\mathrm{TorrGen}(\mathcal{D},\sigma)$ with highest probability.

Definition 7.1. Let $\mathcal{D}$ be an MC, and $\sigma \in \mathrm{Paths}^\star(\mathrm{Ac}(\mathcal{D}_\psi))$ a rail of $\mathcal{D}$. We define the *representant of* $\mathrm{Torr}(\mathcal{D},\sigma)$ as

$$\mathrm{repTorr}\,(\mathcal{D},\sigma) = \mathrm{repTorr}\left(\biguplus_{\rho\in\mathrm{TorrGen}(\mathcal{D},\sigma)} \langle\rho\rangle\right) \triangleq \arg\max_{\rho\in\mathrm{TorrGen}(\mathcal{D},\sigma)} \mathbf{Pr}(\langle\rho\rangle)$$

Note that given $\mathrm{repTorr}\,(\mathcal{D},\sigma)$ one can easily recover σ. Therefore, no information is lost by presenting torrents as one of its generators instead of as a rail.

Expanding SCC. Note that in the Preprocessing stage, we reduced the size of many SCCs of the system (and likely even completely removed some) by making states in $\mathrm{Sat}(\psi) \cup S \setminus \mathrm{Sat}_\Diamond(\psi)$ absorbing. However, It is possible that the system still contains some very large strongly connected components. In that case, a single witness could have a very large probability mass and one could argue that the information presented to the user is not detailed enough. For instance, consider the Markov chain of Figure 6 in which there is a single large SCC with input state t and output state u.

The most indicative torrent-counterexample to the property $\mathcal{D} \models_{\leq 0.9} \Diamond\psi$ is simply {TorrGen(stu)}, i.e., a single witness with probability mass 1 associated to the rail stu. Although this may seem uninformative, we argue that it is more informative than listing several paths of the form $st \cdots u$ with probability summing up to, say, 0.91. Our single witness counterexample suggests that the outgoing transition to a state not reaching ψ was simply forgotten in the design; the listing of paths still allows the possibility that one of the probabilities in the whole system is simply wrong.

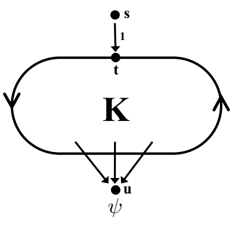


Fig. 6

Nevertheless, if the user needs more information to tackle bugs inside SCCs, note that there is more information available at this point. In particular, for every strongly connected component K, every input state s of K (even for every state in K), and every output state t of K, the probability of reaching t from s is already available from the computation of Ac($\mathcal{D}_\psi$) during the SCC analysis stage of Section 7.2.

8 Final Discussion

We have presented a novel technique for representing and computing counterexamples for nondeterministic and probabilistic systems. We partition a counterexample in witnesses and state five properties that we consider valuable in order to increase the utility of witnesses as a debugging tool: (similarity) elements of a witness should provide similar debugging information; (originality) different witnesses should provide different debugging information; (accuracy) witnesses with higher probability should indicate system behavior more likely to contain errors; (significance) probability of a witness should be relatively high; (finiteness) there should be finitely many witnesses. We achieve this by grouping finite paths in a counterexample together in a witness if they behave the same outside the strongly connected components.

Presently, some work has been done on counterexample generation techniques for different variants of probabilistic models (Discrete Markov chains and Continues Markov chains) [AHL05, AL06, HK07a, HK07b]. In our terminology, these works consider witnesses consisting of a *single* finite path. We have already discussed in the Introduction that the single path approach does not meet the properties of accuracy, originality, significance, and finiteness.

Instead, our witness/torrent approach provides a high level of abstraction of a counterexample. By grouping together finite paths that behave the same outside strongly connected components in a single witness, we can achieve these properties to a higher extent. Behaving the same outside strongly connected components is a reasonable way of formalizing the concept of providing *similar* debugging information. This grouping also makes witnesses significantly different from each other: each witness comes from a different rail and each rail provides a different way to reach the undesired property. Then each witness provides *original* information. Of course, our witnesses are more *significant* than single finite paths, because they are sets of finite paths. This also gives us more *accuracy* than the approach with single finite paths, as a collection of finite paths behaving the same and reaching an undesired condition with high probability is more likely to show how the system reaches this condition than just a single path. Finally, because there is a finite number of rails, there is also a *finite* number of witnesses.

Another key difference of our work with previous ones is that our technique allows to generate counterexamples for probabilistic systems *with* nondeterminism. However, a recent report [AL07] also considers counterexample generation for MDPs. Their approach only extends to upper bounded pCTL formulas without nested temporal operators. We would like to remark that our technique to approach counterexample generation for MDPs completely differs from theirs.

Finally, we are not aware of any other work in the literature considering counterexamples for probabilistic LTL model checking.

The authors would like to stress the important result of [HK07a], which provides a systematic characterization of counterexample generation in terms of shortest paths problems. We use this result to generate counterexamples for the acyclic Markov chains.

In the future we intend to implement a tool to generate our significant diagnostic counterexamples; a very preliminary version has already been implemented. There is still work to be done on improving the visualization of the witnesses, in particular, when a witness captures a large strongly connected component. Another direction is to investigate how this work can be extended to timed systems, either modeled with continuous time Markov chains or with probabilistic timed automata.

Acknowledgement. The authors thank David Jansen for helpful comments on an earlier version of this paper.

References

[AD06] Andrés, M.E., D'Argenio, P.: Derivation of counterexamples for quantitative model checking. Master's thesis, Universidad Nacional de Córdoba (2006)

[AHL05] Aljazzar, H., Hermanns, H., Leue, S.: Counterexamples for timed probabilistic reachability. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 177–195. Springer, Heidelberg (2005)

[AL06] Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 33–51. Springer, Heidelberg (2006)

[AL07] Aljazzar, H., Leue, S.: Counterexamples for model checking of markov decision processes. Computer Science Technical Report soft-08-01, University of Konstanz (December 2007)

[Alf97] De Alfaro, L.: Temporal logics for the specification of performance and reliability, pp. 165–176. Springer, Heidelberg (1997)

[BdA95] Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.) Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1995), vol. 1026, pp. 499–513 (1995)

[Bel57] Bellman, R.E.: A Markovian decision process. J. Math. Mech. 6, 679–684 (1957)

[BLR05] Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Perform. Eval. Rev. 32(4), 34–40 (2005)

[Cas93] Cassandras, C.G.: Discrete Event Systems: Modeling and Performance Analysis. Richard D. Irwin, Inc./Aksen Associates, Inc. (1993)

[CGJ$^+$00] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification, pp. 154–169 (2000)

[dA97] de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D thesis, Stanford University (1997)

[Epp98] Eppstein, D.: Finding the k shortest paths. SIAM Journal of Computing, 652–673 (1998)

[FV97] Filar, J., Vrieze, K.: Competitive Markov Decision Processes (1997)

[HK07a] Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)

[HK07b] Han, T., Katoen, J.-P.: Providing evidence of likely being on time–counterexample generation for ctmc model checking. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 331–346. Springer, Heidelberg (2007)

[MP91] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1991)

[PZ93] Pnueli, A., Zuck, L.D.: Probabilistic verification. Information and Computation 103(1), 1–29 (1993)

[SdV04] Sokolova, A., de Vink, E.P.: Probabilistic automata: System types, parallel composition and comparison. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 1–43. Springer, Heidelberg (2004)

[SL95] Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995)

[Var85] Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state systems. In: Proc. 26th IEEE Symp. Found. Comp. Sci., pp. 327–338 (1985)

Statistical Model Checking of Mixed-Analog Circuits with an Application to a Third Order $\Delta - \Sigma$ Modulator[⋆]

Edmund Clarke, Alexandre Donzé, and Axel Legay

School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213
{emc,adonze,alegay}@cs.cmu.edu

Abstract. In this paper, we consider verifying properties of mixed-signal circuits, i.e., circuits for which there is an interaction between analog (continuous) and digital (discrete) quantities. We follow the statistical Model Checking approach of [You05, You06] that consists of evaluating the property on a representative subset of behaviors, generated by simulation, and answering the question of whether the circuit satisfies the property with a probability greater than or equal to some value. The answer is correct up to a certain probability of error, which is prespecified. The method automatically determines the minimal number of simulations needed to achieve the desired accuracy, thus providing a convenient way to control the trade-off between precision and computational cost. We propose a logic adapted to the specification of properties of mixed-signal circuits, in the temporal domain as well as in the frequency domain. Our logic is unique in that it allows us to compare the Fourier transform of two signals. We demonstrate the applicability of the method on a model of a third order $\Delta - \Sigma$ modulator for which previous formal verification attempts were too conservative and required excessive computation time.

1 Introduction

Given a property ϕ, the *Probabilistic Model Checking Problem* consists of checking whether a stochastic system satisfies ϕ with a probability greater than or equal to a certain threshold θ. This problem is generally solved with a *numerical approach* that consists of computing the *exact* probability for the system to satisfy ϕ and by comparing the result to θ. The way the probability is computed

⋆ This research was sponsored by the GSRC (University of California) under contract no. SA423679952, National Science Foundation under contracts no. CCF0429120, no. CNS0411152, and no. CCF0541245, Semiconductor Research Corporation under contract no. 2005TJ1366, Air Force (University of Vanderbilt) under contract no. 18727S3, International Collaboration for Advanced Security Technology of the National Science Council, Taiwan, under contract no. 1010717, and a grant from the Belgian American Educational Foundation.

depends on the nature of the system as well as on the property that is considered. Successful results (see e.g. [BHHK03, CY95, CG04]) and tools (see e.g. [KNP04, CB06]) exist for various classes of systems, including (continuous time) Markov Chains and Markov Decision Processes. The drawback behind numerical approaches is that they compute the probability by considering all the executions of the system, which may not scale up for systems of large size. Another way to solve the probabilistic Model Checking problem is to use a *statistical approach* based on hypothesis testing and simulation (e.g., [You05, You06] or [SVA04, SVA05]). The key idea is to deduce whether or not the system satisfies the property by observing some of its executions. Of course, in contrast to a numerical approach, a test-based solution does not guarantee a correct result. However, it is possible to bound the probability of making an error. Statistical approaches are known to be far less memory and time intensive than numerical ones, and are sometimes the last resort [YKNP06].

In this paper, we consider applying the statistical procedure proposed by Younes in [You05, You06] to verify properties of *mixed-signal circuits*, i.e., circuits for which there is an interaction between analog (continuous) and digital (discrete) quantities. Our first contribution is to propose a version of stochastic discrete-time event systems that fits into the framework of [You05, You06] with the additional advantage that it explicitly handles analog and digital signals. We also introduce *probabilistic signal linear temporal logic*, a logic adapted to the specification of properties for mixed-signal circuits in the *temporal* domain and in the *frequency* domain.

Our second contribution is the analysis of a $\Delta-\Sigma$ modulator. A $\Delta-\Sigma$ modulator is an efficient *Analog-to-Digital Converter circuit*, i.e., a device that converts analog signals into digital signals. A common critical issue in this domain is the analysis of the *stability* of the internal state variables of the circuit. The concern is that the values that are stored by these variables can grow out of control until reaching a maximum value, causing the circuit to *saturate*. Saturation is commonly assumed to compromise the quality of the analog-to-digital conversion. In [DDM04] and [GKR04] reachability techniques developed in the area of hybrid systems were used to analyze the stability of a third-order modulator. The idea was to use these techniques to guarantee that for *every* input signal in a given range, the states of the system remain stable. While this reachability-based approach is strictly precise, it has important drawbacks such as (1) signals with long duration cannot be practically analyzed and (2) there are interesting properties that cannot be checked. Our results show that a statistical Model Checking approach makes it possible to handle properties and signals that are beyond the scope of the reachability-based approach. As an example, in our experiments, we have been able to analyze discrete signals with more than 24000 sampling points in seconds, while the approach in [DDM04] was limited to 31 points in hours. We are also able to provide insight on an open question in [DDM04] by observing that saturation does not always imply an improper signal conversion. The latter can be done by comparing the Fourier transform of each of the input analog signals with the Fourier transform of its corresponding digital signal. Such a property can easily be expressed in our logic and model checked with our statistical-based approach. We are unaware of any other formal verification technique that can solve this problem.

2 Statistical Probabilistic Model Checking

The following section introduces the technique of *Statistical Probabilistic Model Checking.*

2.1 The Probabilistic Model Checking Problem

We use $Pr(E)$ to denote the probability of event E. We consider a stochastic system $\mathcal{S}$ whose executions are *observable* and a property ϕ. We assume that one can decide whether an execution of $\mathcal{S}$, denoted by σ, satisfies ϕ. The *Probabilistic Model Checking Problem* consists of deciding whether the executions of $\mathcal{S}$ satisfy ϕ with a probability greater than or equal to a given threshold θ. The latter is denoted by $\mathcal{S} \models Pr_{\geq\theta}(\phi)$. This problem is well-defined if and only if one can assign a probability to the set of executions of $\mathcal{S}$ that satisfy ϕ. One way to solve the Probabilistic Model Checking Problem is to use a numerical approach (see the introduction). The drawback with such an approach is that it computes the probability for all the executions of the system and may not scale up for systems of large size. Another way to solve the probabilistic Model Checking problem is to use a *statistical model checking algorithm.* In the rest of this section, we recap the statistical Model Checking technique proposed by Younes in [You05, You06].

2.2 Statistical Approach

The approach in [You05, You06] is based on hypothesis testing. The idea is to check the property ϕ on a sample set of simulations and to decide whether the system satisfies $Pr_{\geq\theta}(\phi)$ based on the number of executions for which ϕ holds compared to the total number of executions in the sample set. With such an approach, we do not need to consider all the executions of the system. To determine whether $\mathcal{S}$ satisfies ϕ with a probability $p \geq \theta$, we can test the hypothesis $H : p \geq \theta$ against $K : p < \theta$. A test-based solution does not guarantee a correct result but it is possible to bound the probability of making an error. The *strength* (α, β) of a test is determined by two parameters, α and β, such that the probability of accepting K (respectively, H) when H (respectively, K) holds, called a Type-I error (respectively, a Type-II error) is less or equal to α (respectively, β).

A test has *ideal performance* if the probability of the Type-I error (respectively, Type-II error) is exactly α (respectively, β). However, these requirements make it impossible to ensure a low probability for both types of errors simultaneously (see [You05] for details). A solution to this problem is to relax the test by working with an *indifference region* (p_1, p_0) with $p_0 \geq p_1$ ($p_0 - p_1$ is the *size of the region*). In this context, we test the hypothesis $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$ instead of H against K. If the value of p is between p_1 and p_0 (the indifference region), then we say that the probability is sufficiently close to θ so that we are indifferent with respect to which of the two hypotheses K or H is accepted. The threshold p_0 and p_1 are generally defined in term of the single threshold θ, e.g., $p_1 = \theta - \delta$ and $p_0 = \theta + \delta$.

2.3 An Algorithmic Scheme

Younes proposed a procedure to test $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$ that is based on the *sequential probability ratio test* proposed by Wald [Wal45]. The approach is briefly described below.

Let B_i be a discrete random variable with a Bernoulli distribution. Such a variable can only take 2 values 0 and 1 with $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$. In our context, each variable B_i is associated with one simulation of the system. The outcome for B_i, denoted b_i, is 1 if the simulation satisfies ϕ and 0 otherwise. In the sequential probability ratio test, one has to choose two values A and B, with $A > B$. These two values should be chosen to ensure that the strength of the test is respected. Let m be the number of observations that have been made so far. The test is based on the following quotient:

$$\frac{p_{1m}}{p_{0m}} = \prod_{i=1}^{m} \frac{Pr(B_i = b_i \mid p = p_1)}{Pr(B_i = b_i \mid p = p_0)} = \frac{p_1^{d_m}(1-p_1)^{m-d_m}}{p_0^{d_m}(1-p_0)^{m-d_m}}, \tag{1}$$

where $d_m = \sum_{i=1}^{m} b_i$. The idea behind the test is to accept H_0 if $\frac{p_{1m}}{p_{0m}} \geq A$, and H_1 if $\frac{p_{1m}}{p_{0m}} \leq B$. An algorithm for sequential ratio testing consists of computing $\frac{p_{1m}}{p_{0m}}$ for successive values of m until either H_0 or H_1 is satisfied. This has the advantage of minimizing the number of simulations. In each step i, the algorithm has to check the property on a single execution of the system, which is handled with a new Bernoulli variable B_i whose realization is b_i. In his thesis [You05], Younes proposed a logarithmic based algorithm (Algorithm 2.3 page 27) SPRT that given p_0, p_1, α and β implements the sequential ratio testing procedure. Computing ideal values A_{id} and B_{id} for A and B in order to make sure that we are working with a test of strength (α, β) is a laborious procedure (see Section 3.4 of [Wal45]). In his seminal paper [Wal45], Wald showed that if one defines $A_{id} \geq A = \frac{(1-\beta)}{\alpha}$ and $B_{id} \leq B = \frac{\beta}{(1-\alpha)}$, then we obtain a new test whose strength is (α', β'), but such that $\alpha' + \beta' \leq \alpha + \beta$, meaning that either $\alpha' \leq \alpha$ or $\beta' \leq \beta$. In practice, we often find that both inequalities hold.

The SPRT algorithm can be extended to handle Boolean combinations of probabilistic properties as well as much more complicated probabilistic Model checking problems than the one considered in this paper [You05].

3 Signals, Systems and Logics

3.1 Signals Definition

We use $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{C}$ to denote the sets of natural, real, and complex numbers, respectively. Let the *time set* $\mathcal{T}$ be a finite set of non-negative reals $\{t_0, t_1, \ldots, t_{N-1}\}$, where $N \in \mathbb{N}$. To simplify the presentation, we assume that $t_{i+1} - t_i = \delta t$, where $\delta t \in \mathbb{R}_{>0}$. A *digital set* is a set consisting of 2^b elements, which can be encoded in terms of b bits. A *frequency set* is a subset of $\mathbb{R}$. An *analog signal* is a mapping $\xi : \mathcal{T} \to \mathbb{R}$. A *digital signal* is a mapping $\xi : \mathcal{T} \to \mathcal{D}$, where $\mathcal{D}$ is a digital set. A *frequency-domain* signal is a mapping $\hat{\xi} : \mathcal{F} \to \mathbb{C}$, where $\mathcal{F}$ is a frequency set. The value at time $t \in \mathcal{T}$ of a signal ξ is denoted by $\xi[t]$. Let $t, t' \in \mathcal{T}$, the *restriction* of a signal ξ to $[t, t']$, denoted by $\xi_{|_{[t,t']}}$, is a signal such that:

$$\xi_{|_{[t,t']}}[\tau] = \begin{cases} \xi[\tau] & \text{if } \tau \in [t, t'] \\ 0 & \text{else.} \end{cases}$$

The restriction of a frequency-domain signal to an interval of frequencies is defined similarly.

The *Fourier transform* (see [Smi97]) is a functional F that maps a time-domain signal $\xi : \mathcal{T} \to \mathbb{R}$ to a *frequency-domain* signal $\hat{\xi} = F(\xi)$. The *inverse Fourier transform* is used to "reconstruct" ξ from $\hat{\xi}$, i.e., $\xi = F^{-1}(\hat{\xi})$. Formally, for all ν in $\mathcal{F}$ and for all t in $\mathcal{T}$ we have

$$F(\xi)[\nu] = \int_{\mathcal{T}} \xi[t] e^{-\mathbf{i}2\pi\nu t} dt \quad \text{and} \quad F^{-1}(\hat{\xi})[t] = \xi[t] = \int_{\mathcal{F}} \hat{\xi}[\nu] e^{\mathbf{i}2\pi\nu t} d\nu.$$

An efficient algorithm known as the *Fast Fourier Transform algorithm* (see, e.g., [FJ97]) is used to compute a discrete approximation of the Fourier transform.

3.2 Model

Our main motivation is to verify properties of mixed-signal circuits. For this purpose, we define *stochastic signal discrete-time event systems*, which extend the classical stochastic discrete-time event systems with information about signals. During an execution, these systems have to remain in the same state between the occurrence of two events. The signals associated with each execution are thus piecewise-constant.

Definition 1. *Let $\mathcal{B}$ be a finite set of Boolean propositions. A* stochastic signal discrete-time event system *(SSDES) is a tuple $\mathcal{S} = (\mathcal{T}, S, s_0, \to, \pi_a, \pi_d, L)$ where*

- *$\mathcal{T}$ is a finite set of non-negative reals $\{t_0, t_1, \ldots, t_{N-1}\}$, with $t_{i+1} - t_i = \delta t$;*
- *S is the set of states, defined as $S = A_s \times D_s$, where $A_s \subset \mathbb{R}^{n_a}$ and $D_s \subset \mathcal{D}^{n_d}$, n_a and n_d being the number of analog and digital signals associated with $\mathcal{S}$, respectively. These signals will be denoted by $\xi_a^1, \ldots, \xi_a^{n_a}$ and $\xi_d^1, \ldots, \xi_d^{n_d}$;*

- *$s_0 \in S$ is the initial state;*
- *The relation $\rightarrow$: $S \times S$ is the transition relation of the system. We assume a probability distribution on $\rightarrow$, i.e.,*

$$\forall s \in S, \sum_{s' \in S} Pr(s \rightarrow s') = 1;$$

  Our model is assumed to have the Markovian property;
- *$\pi_a : S \times \{1, \ldots, n_a\} \rightarrow A_s$ is a* projection operator *such that for all $s = (s_a^1, \ldots, s_a^{n_a}, s_d^1, \ldots, s_d^{n_d})$ and $1 \leq j \leq n_a$, $\pi_a(s, j) = s_a^j$;*
- *π_d is defined in a similar manner to π_a.*
- *L is a mapping from S to $2^{\mathcal{B}}$, which assigns to each state the elements in $\mathcal{B}$ that are true in that state. If $p \in L(s)$, then we say that s satisfies p.*

Let $\omega = s_1 \ldots s_k$ be a finite sequence of states of $\mathcal{S}$. We use $\omega(i)$ and ω^i to denote the i-th state of ω and the sequence $s_i \ldots s_k$, respectively. The length ω, denoted $|\omega|$, is the number of states in ω. An *execution* of an SSDES $\mathcal{S} = (\mathcal{T}, S, s_0, \rightarrow, \pi_a, \pi_d, L)$ is a sequence of N states $\sigma = s_0 s_1 \ldots s_{N-1}$ such that for each $i \in 0 \ldots N-1$, $s_i \in S$ and $s_i \rightarrow s_{i+1}$. Each state s_k (with $k < N$) of σ assigns to each analog signal ξ_a^i (respectively, digital signal ξ_d^i) its constant value between t_k and t_{k+1}, i.e., $\xi_a^i[t] = \pi_a(s_k, i)$ (respectively, $\xi_d^i[t] = \pi_d(s_k, i)$) for $t \in [t_k, t_{k+1}]$. The i-th *suffix* of σ is the sequence $s_i, \ldots, s_{N-1}$. An SSDES is thus an infinite-state Markov Chain equipped with information and operations on analog and digital signals.

3.3 Probabilistic Signal Linear Temporal Logic

We introduce the *probabilistic signal linear temporal logic* (SLTL) to reason on the set of executions of an SSDES. In the rest of the section, we assume a set of atomic propositions $\mathcal{B}$ and an SSDES $\mathcal{S} = (\mathcal{T}, S, S_0, \rightarrow, \pi_a, \pi_d, L)$ with L being a mapping from the set of states S to $2^{\mathcal{B}}$. Before introducing SLTL, we first recall the syntax and the semantics for linear temporal logic (LTL). The syntax of LTL is given by the following grammar:

$$\phi ::= \mathbf{T} \,|\, \mathbf{F} \,|\, b \in \mathcal{B} \,|\, \phi_1 \vee \phi_2 \,|\, \phi_1 \wedge \phi_2 \,|\, \neg\phi \,|\, \bigcirc \phi \,|\, \phi_1 \mathcal{U} \phi_2 \,|\, \phi_1 \widetilde{\mathcal{U}} \phi_2.$$

We now present the semantics of LTL, which here is defined with respect to finite sequences of states of $\mathcal{S}$. The fact that a finite sequence of states ω of $\mathcal{S}$ satisfies the LTL property ϕ is denoted by $\omega \models \phi$. We have the following:

- $\omega \models \mathbf{T}$ and $\omega \not\models \mathbf{F}$;
- $\omega \models b$ with $b \in \mathcal{B}$ if and only if $b \in L(\omega(0))$;
- $\omega \models \phi_1 \vee \phi_2$ if and only if $\omega \models \phi_1$ or $\omega \models \phi_2$;
- $\omega \models \phi_1 \wedge \phi_2$ if and only if $\omega \models \phi_1$ and $\omega \models \phi_2$;
- $\omega \models \neg\phi$ if and only if $\omega \not\models \phi$.
- $\omega \models \bigcirc\phi$ if and only if $|\omega| > 1$ and $\omega^1 \models \phi$;
- $\omega \models \phi_1 \mathcal{U} \phi_2$ if and only if there exists $0 \leq i \leq |\omega| - 1$ such that $\omega^i \models \phi_2$, and for each $0 \leq j < i$, $\omega^j \models \phi_1$;

– $\omega \models \phi_1 \widetilde{\mathcal{U}} \phi_2$ if and only if for each $0 \leq i \leq |\omega| - 1$ such that $\omega^i \not\models \phi_2$ there exists $0 \leq j < i$ such that $\omega^j \models \phi_1$;

Two additional temporal operators are used, that are $\Diamond\psi = \mathbf{T}\mathcal{U}\psi$ and $\Box\psi = \mathbf{F}\widetilde{\mathcal{U}}\psi$.

Note that we consider LTL properties on finite executions. Thus, we can only specify bounded LTL properties. As in [You05], we thus stay in the class of safety properties. It is easy to decide whether a finite execution satisfies a LTL formula. We now introduce the notion of an *execution predicate*.

Definition 2 (Execution Predicate). *Let $\Sigma(\mathcal{S})$ be the set of all the executions of an SSDES $\mathcal{S}$. An* execution predicate *p for $\mathcal{S}$ is a predicate on $\Sigma(\mathcal{S})$.*

Example 1. Consider an execution predicate p that decides whether the mean value of the first analog signal associated with an execution σ of an SSDES is greater than 0. Such predicate can be defined as

$$p(\sigma) = \mathbf{T} \quad \text{iff} \quad \frac{1}{N}\sum_{k=0}^{N-1} \pi_a(\sigma(k), 1) \geq 0.$$

This example shows that the definition of execution predicate makes it easy to define properties on entire executions that cannot easily be defined with temporal operators. In Section 5, we will consider a more complex execution predicate that compares the Fourier transforms of two signals.We add a new clause to the grammar of LTL for execution predicates. Let $\mathcal{P}$ be a set of execution predicates, our new grammar for LTL is

$$\phi ::= \mathbf{T} \,|\, \mathbf{F} \,|\, p \in \mathcal{P} \,|\, b \in \mathcal{B} \,|\, \phi_1 \vee \phi_2 \,|\, \phi_1 \wedge \phi_2 \,|\, \neg\phi \,|\, \bigcirc \phi \,|\, \phi_1 \mathcal{U} \phi_2 \,|\, \phi_1 \widetilde{\mathcal{U}} \phi_2,$$

with the restriction that execution predicates cannot be under the scope of temporal operators. We can now define probabilistic signal linear temporal logic.

Definition 3 (SLTL Formula). *An* SLTL formula *is a formula of the form $\psi = Pr_{\geq\theta}(\phi)$, where ϕ is a LTL formula with execution predicates.*

We say that $\mathcal{S}$ satisfies ψ, denoted by $\mathcal{S} \models \psi$ if and only if the probability for an execution of $\mathcal{S}$ to satisfy ϕ is greater or equal than θ. The problem is well-defined since, as is shown in the following theorem, one can always assign a unique probability measure to the set of executions that satisfy an LTL formula with execution predicates.

Theorem 1. *Let $\mathcal{S}$ be an SSDES and ϕ be a LTL property with execution predicates. One can always associate a unique probability measure to the set of executions of $\mathcal{S}$ that satisfy ϕ.*

The proof can be found in [EC08].

Assuming that we are only working with execution predicates that we can compute, we observe that SSDES and SLTL are in the scope of the class of systems and logics that can be handled with the SPRT algorithm.

4 A Class of Mixed-Signal Circuits: $\Delta - \Sigma$ Modulators

This section is a brief introduction to the principles of $\Delta - \Sigma$ *modulation* and the related design issues. The reader can consult [MPVRV01] for more details on this topic in Signal Processing.

4.1 Analog to Digital Conversion via $\Delta - \Sigma$ Modulation

A $\Delta - \Sigma$ modulator is an *Analog-to-Digital Converter circuit*, i.e., a circuit that takes an analog value $u \in \mathbb{R}$ as input and encodes it into a digital value $v \in \mathcal{D}$. Since digital signal processing is more widely used than analog signal processing, such converters are found in many electrical devices, which motivates their study. The challenge with Analog-to-Digital conversion is to represent the uncountable set of analog values using a finite set of digital values $\mathcal{D}$. The direct approach, which is called *quantization*, consists in mapping u to the digital value v that minimizes the *quantization error* defined as $\delta = u - v$, i.e., it chooses $v = \text{argmin}_{v \in \mathcal{D}} |\delta|$. Obviously, one way to decrease the remaining quantization error is to increase the number of bits used to encode $\mathcal{D}$ and thus the number of possible digital values. Another approach, which is implemented by $\Delta - \Sigma$ modulation, is to measure and compensate for the accumulation of quantization errors during time. As an example, consider the following simple instance of a discrete time $\Delta - \Sigma$ modulator. Let $u(k)$, $v(k)$, $\delta(k) = u(k) - v(k)$ be the analog input, the digital output, and the quantization error at step k, respectively. The modulator uses an *integrator* to store the accumulation of errors in a variable $x(k) = \sum_0^k \delta(k)$, so that $x(k+1) = x(k) + \delta(k)$, and determines the next digital output $v(k+1)$ based on the sign of $x(k+1)$, i.e., $\mathcal{D} = \{-1, 1\}$ and $v(k+1) = 1$ if $x(k+1) \geq 0$ and $v(k+1) = -1$ otherwise. A $\Delta - \Sigma$ modulator thus basically consists of a feedback loop controlling the quantization error. To improve the performance, more complex feedback loops can be designed involving more than one integrator. The *order* of a modulator is given by the number of integrators used.

The benefit of the $\Delta - \Sigma$ modulation approach is clearly apparent in the frequency domain. Indeed, the Fourier transform of the digital signal is the Fourier transform of the analog signal composed with some error due to the quantization. The feedback loop in the $\Delta - \Sigma$ modulator is designed to "push" this error towards high frequencies, where it can be isolated and removed, e.g. by using a low-pass filter (see Fig. 2). The original signal can then be retrieved by using the inverse Fourier transform.

4.2 Verification Issues

Modulators with more than two integrators are known to exhibit better performance but also introduce a *stability* issue [ASS96]. An integrator memorizes its input and adds it to the sum of all the previously read inputs during the execution. Consequently, an important issue is whether the integrators are stable, i.e., whether or not the values stored in the integrators can grow indefinitely.

Because integrators have limited capacity, the values of these states would then reach a *saturation level*. Saturation can compromise the quality of the analog-to-digital conversion. The stability analysis of the feedback loop is made difficult by the nonlinearity (in this case, a discontinuity) induced by quantization. This invalidates the direct application of classical linear stability theory which makes the stability analysis of $\Delta - \Sigma$ modulators a challenging problem (see [SH93]). In the next section, we investigate several issues related to stability by using a statistical Model Checking approach.

5 Experimental Results

We implemented a prototype in the MATLAB environment. Our procedure takes as input a Simulink model and a property ϕ that is a LTL formula with execution predicates. To apply our statistical approach, we combine the Simulink model with a stochastic input generator. At each time instant t_i, this generator randomly chooses an input value for the analog signal and the Simulink engine uses this value to compute the next state of the system. The result is an SSDES whose executions can easily be observed without building the entire state-space of the system. We now discuss the experimental results we obtained when applying our prototype to a third-order $\Delta - \Sigma$ modulator.

5.1 SSDES for a Third Order Modulator

We work with the instance of a third order $\Delta - \Sigma$ modulator that was considered in [DDM04]. A Simulink model is given in Fig. 1. It is combined with a stochastic input generator to give an SSDES $\mathcal{S}=(\mathcal{T}, S, s_0, \rightarrow, \pi_a, \pi_d, L)$, where

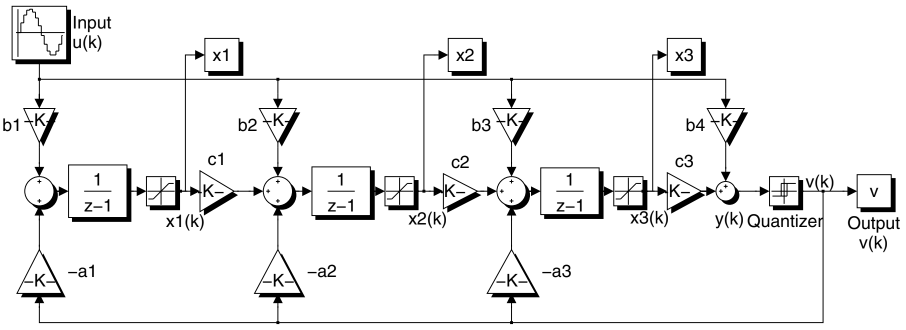


Fig. 1. Simulink model of a third order $\Delta - \Sigma$ modulator.The three blocks $\frac{1}{z-1}$ followed by saturation blocks represent the saturated integrators. The values of the coefficients a_i, b_i and c_i were obtained using the `delsig` toolbox. They are $a_1 = b_1 = 0.0440$, $a_2 = b_2 = 0.2881$, $a_3 = b_3 = 0.7997$, $b_4 = 1$, and $c_1 = c_2 = c_3 = 1$; x_1, x_2 and x_3 are the analog variables storing the integrators states.

- **Time.** We set $\mathcal{T} = \{t_0, t_1, .., t_{N-1}\}$ with $t_0 = 0, t_{N-1} = 3$ and $\delta t = t_{i+1} - t_i = \frac{1}{8000}$, $N = 24000$.

- **Set of States.** The Simulink model contains three integrators such that each contains one real-valued (or analog) variable. A state $s \in S$ can thus be described as a tuple (u, x_1, x_2, x_3, v), where
  - x_1, x_2 and x_3 are analog variables storing the integrators' states;
  - u is an analog variable storing values for the input signal ξ^u;
  - v is a digital variable storing values for the output signal ξ^v.

  The number of analog signals is thus $n_a = 4$ and the number of digital signals $n_d = 1$. We assume that the states of the integrators cannot go beyond certain values that are fixed by the model. When this value is reached, we say that the integrators saturate. In practice, $x_i \in [-1, 1]$ for $i \in \{1, 2, 3\}$ and $-1, 1$ are the *saturation values.* Assuming also that $u \in [-u_{\max}, u_{\max}]$, we get $A_s = [-1, 1]^3 \times [-u_{\max}, u_{\max}]$ and $D_s = \{-1, 1\}$. Given an execution $\sigma = s_0 s_1 \dots s_{N-1}$, we use $u(k) = \pi_a(s_k, 1)$, $x_1(k) = \pi_a(s_k, 2)$, $x_2(k) = \pi_a(s_k, 3)$, $x_3(k) = \pi_a(s_k, 1)$ and $v(k) = \pi_d(s_k, 1)$. For all $k \in \{0, \dots, N-1\}$, we have $\xi^u[t_k] = u(k)$ and $\xi^v[t_k] = v(k)$;

- **Transition relation.** When $u(k)$ is given, the Simulink engine computes $x_1(k+1)$, $x_2(k+1)$, $x_3(k+1)$ and $v(k+1)$. Thus the probability distribution $Pr(s_k \to s_{k+1})$ for all $(s_k, s_{k+1}) \in S \times S$ is induced by the probability distribution of the input value $u(k+1)$. For our experiments, we consider uniform random inputs: for all k, $u(k)$ is chosen in a set $[-u_{\max}, u_{\max}]$ with a uniform random distribution;

- **Initial state.** Initially, the values of the integrator states are 0 and by convention the digital output $v(0)$ is set to 1 and the input value $u(0)$ to 0 Thus the initial state is $s_0 = (0, 0, 0, 0, 1)$;

- **Boolean variables.** We define a Boolean variable $Satur$ which is true iff one of the analog values, i.e., either the input or an integrator state, saturates. Formally, $L(s) = \mathbf{T}$ iff there exist i in $\{1, \dots, 4\}$ such that $\pi_a(s, i) = 1$ or -1, $L(s) = \mathbf{F}$ otherwise.

The choice of the probability distribution to generate input signals influences the statistical result we obtain. A simple choice is the *uniform* distribution, which gives the same probability for every possible input signal to occur. By doing so, we make as few assumptions as possible on the nature of the input signal. We can thus compare our results with those obtained on the corresponding non-stochastic model.

5.2 Experiments

Saturation We first considered the formula $Pr_{\geq\theta}(\Diamond Satur)$, i.e., whether saturation occurs with a probability greater or equal to θ for different values of

Table 1. Table of results for $Pr_{\geq\theta}(\Diamond Satur)$. H_0 was rejected for the first line and accepted for the others.

$u_{\max}$	Probability θ checked	Number of exec.
0.1	0	416
0.15	0.09	4967
0.2	0.64	17815
0.25	0.98	416
0.3	1	688

$u_{\max}$. We applied the SPRT algorithm for several values[1] of θ. We set the two error bounds α and β to 0.001 and used an indifference region $(p_1, p_0) = (\theta - 0.01, \theta + 0.01)$. We tested $H_0 : p \geq \theta + 0.01$ against $H_1 : p \leq \theta - 0.01$. Our results are reported in Table 2. The first and second column report the value of $u_{\max}$ and the value of θ chosen, respectively. Column 3 reports the number of simulations performed. H_0 was rejected for the first line and accepted for the others. Our results show that saturation will occur with probability 1 when the maximum amplitude $u_{\max}$ of the input signal is greater than 0.3.

In [DDM04] and [GKR04] reachability techniques developed in the area of hybrid systems were used to guarantee that for *every* input signal in a given range, the integrator state will never saturate. While this approach is clearly sound for proving stability, its computational cost is prohibitive. As an example, in [DDM04], stability was only proved for a small number of steps, i.e., $N = 31$. Our results can be compared with those reported[2] in [DDM04]. In particular, we confirmed the fact that for signals with a maximum amplitude of 0.1, the circuit never saturates whereas if $u_{\max}$ is more than 0.3, the circuit always does. In our case, though, the length N of the executions considered was much larger.

Frequency Domain Predicate. In addition to improving the computation time, our approach makes it possible to verify more complex properties than those that can be handled with a reachability-based technique. In particular, by defining execution predicates involving the Fourier transform, we can check reliably whether an analog signal was properly converted to a digital one. We can also investigate the relation between saturation and wrong behaviors of the modulator without assuming a priori, as is the case in [DDM04], that the latter implies the former. We checked the formula $Pr_{\geq\theta}(p_F)$, where p_F is a frequency-domain execution predicate that compares the Fourier transform of the input analog signal u with the one of its corresponding digital signal v. Formally, p_F

[1] The values for $u_{\max} = 0.1$ and $u_{\max} = 0.3$ were chosen to validate the experiments in [DDM04], while the others were chosen with some trial and error process to get closer to true probability.

[2] Recall that the results in [DDM04] are obtained from the Simulink model, while we work with the corresponding stochastic model.

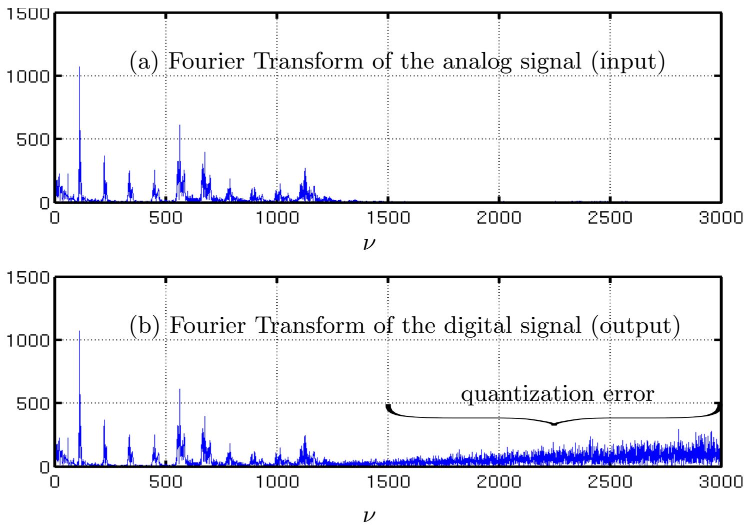


Fig. 2. A sample behavior of the $\Delta - \Sigma$ modulator. The Fourier transform of the output signal (b) matches the Fourier transform of the input signal (a) on the interval $[0, 1500Hz]$. The quantization error is pushed toward frequencies higher than $1500Hz$.

is defined as follows. Let d_F be a metric on frequency-domain signals such that for two signals $\hat{\xi}_1$ and $\hat{\xi}_2$,

$$d_F(\hat{\xi}_1, \hat{\xi}_2) = \frac{1}{N} \sum_{0 \le k \le N-1} |\hat{\xi}_1[\nu_k] - \hat{\xi}_2[\nu_k]|. \tag{2}$$

Let σ be an execution of $\mathcal{S}$. The value of the execution predicate p_F on σ is given by $p_F(\sigma) = \mathbf{T}$ iff $d_F(\hat{\xi}^u_{|[0,\nu]}, \hat{\xi}^v_{|[0,\nu]}) \le \epsilon$, where $\hat{\xi}^u$ and $\hat{\xi}^v$ are the Fourier transforms of the input analog signal u and its corresponding digital signal v, respectively. It is easy to derive a MATLAB routine that can decide whether or not an execution provided by Simulink satisfies p_F. We worked with $\nu = 100Hz$ and $\epsilon = 0.1$, since we observed that for those values the predicate efficiently discriminates between executions for which the digital output has a correct Fourier transform (see Figure 2) against executions when this is not the case). We used the same indifference region and the same error types as in the previous experiments.

In our experiments (reported in Table 2), we observed that p_F is true with probability $\geq$1 for $u_{\max} = 0.8$ and that this probability decreases when the value of $u_{\max}$ increases[3]. This means that saturation does not always imply a wrong behavior. Indeed, as an example, for values of $u_{\max}$ greater than 0.3, the property $Pr_{\geq 1}(\Diamond Satur)$ holds (see previous experiment) and for values of $u_{\max}$ smaller than 0.8, the property $Pr_{\geq 1}(p_F)$ also holds. We can thus infer that between 0.3

[3] We also observed that for values of $u_{\max}$ greater or equal to 0.9, the probability for a good conversion to occur was strictly inferior to 1. The values of θ reported in lines $2 - 5$ of Table 2 have been found with some trial and error process.

Table 2. Table of results for $Pr_{\geq\theta}(p_F)$. H_0 was accepted for each experiment

$u_{\max}$	Probability θ checked	Number of exec.
0.8	1.	688
0.9	0.98	612
1.0	0.98	1248
1.1	0.875	6388
1.2	0.575	15507

and 0.8, the property $\Diamond Satur \wedge p_F$ holds with probability 1. In [DDM04], it is assumed that the absence of saturation is necessary for p_F to be true. Our experiments show that this may be an overly conservative assumption.

More experimental results, in particular characterizing the computation time with respect to the strength parameters α and β, and the size of the indifference region can be found in [EC08].

6 Future Work

This paper presents the first attempt to apply the statistical Model Checking techniques introduced in [You05, YS06] to verifying non-trivial properties of mixed-signal circuits. In comparison to [DDM04], our technique allows us to obtain better performance results as well as to handle a larger class of properties. Our results are correct up to a prespecified probability of an error, while those of [DDM04] are exact.

Our work requires the ability to monitor properties of discrete-time signals, which can easily be done with existing techniques [LS06, dR]. In a series of recent papers [NM07, MNP08], Nickovic et al. proposed techniques for monitoring properties of *dense-time* analog signals. An interesting direction would be to adapt the procedure of Younes to work in this latter, more demanding context.

In our experiments, the choice of the value for θ has been driven by the previous observations reported in [DDM04]. In future work, we plan to use the estimation-based method of [HLMP04] to approximate the value of θ for which the property holds.

We also intend to consider extensions of SLTL incorporating past temporal operators and a better correlation between execution predicates and temporal operators. We plan to define more complex specifications for frequency domain properties based on the needs of designers of mixed signal circuits. Our ultimate goal is to provide them with a general framework for specifying and verifying properties of mixed-signal circuits.

Acknowledgements

We thank H. Younes for answering many email questions on his work and C. J. Langmead for fruitful discussions and proof-reading drafts of the paper. We also thank the participants and the referees of the *Formal Verification of Analog Circuits workshop 2008* (where a preliminary version of this work has been presented) for their helpful comments and suggestions.

References

[ASS96] Aziz, P.M., Sorensen, H.V., Van Der Spiegel, J.: An overview of sigma-delta converters. IEEE Signal Processing Magazine, 61–84 (January 1996)

[BHHK03] Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time markov chains. IEEE Trans. Software Eng. 29(6), 524–541 (2003)

[CB06] Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: QEST, pp. 131–132. IEEE, Los Alamitos (2006)

[CG04] Ciesinski, F., Größer, M.: On probabilistic computation tree logic. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 147–188. Springer, Heidelberg (2004)

[CY95] Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)

[DDM04] Dang, T., Donze, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid systems techniques. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 21–36. Springer, Heidelberg (2004)

[dR] d'Amorim, M., Roşu, G.: Efficient monitoring of ω-languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)

[EC08] Legay Edmund Clarke, A., Donzé, A.: Statistical model checking of mixedanalog circuits. In: Second Workshop on Formal Verification of Analog Circuits of CAV 2008 (July 2008)

[FJ97] Frigo, M., Johnson, S.G.: The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology (September 1997)

[GKR04] Gupta, S., Krogh, B.H., Rutenbar, R.A.: Towards formal verification of analog designs. In: ICCAD, pp. 210–217 (2004)

[HLMP04] Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)

[KNP04] Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 2.0: A tool for probabilistic model checking. In: QEST, pp. 322–323. IEEE, Los Alamitos (2004)

[LS06] Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)

[MNP08] Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 475–505. Springer, Heidelberg (2008)

[MPVRV01] Medeiro, F., Pérez-Verdú, B., Rodríguez-Vázquez, A.: Top-Down Design of High-Performance Sigma-Delta Modulators, ch. 2. Kluwer Academic Publishers, Dordrecht (2001)

[NM07] Nickovic, D., Maler, O.: Amt: A property-based monitoring tool for analog systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007)

[SH93] Zakhor, A., Hein, S.: On the stability of sigma delta modulators. IEEE Transactions on Signal Processing 41 (July 1993)

[Smi97] Smith, S.W.: The scientist and engineer's guide to digital signal processing. California Technical Publishing, San Diego (1997)

[SVA04] Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)

[SVA05] Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)

[Wal45] Wald, A.: Sequential tests of statistical hypotheses. Annals of Mathematical Statistics 16(2), 117–186 (1945)

[YKNP06] Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)

[You05] Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. Ph.D thesis, Carnegie Mellon (2005)

[You06] Younes, H.L.S.: Error control for probabilistic model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 142–156. Springer, Heidelberg (2005)

[YS06] Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Information and Computation 204(9), 1368–1409 (2006)

Structural Contradictions

Cindy Eisner[1] and Dana Fisman[1,2]

[1] IBM Haifa Research Laboratory
[2] Hebrew University

Abstract. We study the relation between *logical contradictions* such as $p \wedge \neg p$ and *structural contradictions* such as $p \cap (p \cdot q)$. Intuitively, we expect the two to be treated similarly, but they are not by PSL, nor by SVA. We provide a solution that treats both kinds of contradictions in a consistent manner. The solution reveals that not all structural contradictions are created equal: we must distinguish between them in order to preserve important characteristics of the logic. A happy result of our solution is that it provides the semantics over the natural alphabet 2^P, as opposed to the current semantics of PSL/SVA that use an inflated alphabet including the cryptic letters $\top$ and $\bot$. We show that the complexity of model checking PSL/SVA is not affected by our proposed semantics.

1 Introduction

A *logical contradiction* is a propositional formula that is not satisfiable, for example $p \wedge \neg p$. In temporal logics such as PSL [8,17] and SVA [18] that contain semi-extended regular expressions, or SEREs, *structural contradictions* arise. A structural contradiction is a SERE that is not satisfiable due to its structure. That is, for any replacement of the propositions in the SERE, the SERE remains unsatisfiable. For example, $p \cap (p \cdot q)$ (the language of words consisting of a single letter on which p holds intersected with the language of words consisting of two letters on which p holds on the first and q on the second) is a structural contradiction, while $(p \cdot q) \cap (p \cdot \neg q)$ is a contradiction, but not a structural one. Structural contradictions were first named in [2], which noted that they were treated differently than logical contradictions by the formal semantics of PSL.

The $\cap$ operator is not the only source of structural contradictions in PSL and SVA. A structural contradiction can also be formed from the fusion operator (denoted here by $\circ$), a kind of overlapping concatenation. The language of SERE $r_1 \circ r_2$ includes words of the form $v_1 \ell v_2$, where v_1 and v_2 are words, ℓ is a letter, $v_1 \ell$ is in the language of r_1, and ℓv_2 is in the language of r_2. The SERE $\lambda \circ p$, where λ denotes the language consisting of the empty word, is a structural contradiction formed without the $\cap$ operator.

Before proceeding, we must first understand a little bit about the use of SEREs in PSL/SVA. The set of SEREs is built from atoms which are propositional formulas using the standard operators concatenation (which we denote $\cdot$), union, intersection and Kleene closure, plus the non-standard fusion operator $\circ$. Let r be a SERE. Then the formula $r!$, a *strong* SERE formula, holds on words containing a non-empty prefix in the language of r. For example:

$$\{a^* \cdot b^* \cdot c\}! \tag{1}$$

holds on a word starting with some number of letters satisfying a followed by some number of letters satisfying b followed by a letter satisfying c, and is equivalent to the LTL formula $[a\ \mathsf{U}\ [b\ \mathsf{U}\ c]]$.

The formula r, a *weak* SERE formula, holds if either $r!$ holds or if the word is "too short" (ends before we have "fallen off" the automaton for r) or "too long" (is an infinite word in which we "got stuck" forever in a starred sub-expression).[1] For example:

$$\{a^* \cdot b^* \cdot c\} \tag{2}$$

holds on words starting with some number of letters satisfying a followed by some number of letters satisfying b followed by a letter satisfying c (in other words, words on which $r!$ holds), on (finite) prefixes of such words (words that are "too short") and also on infinite words consisting of an infinite number of letters satisfying a or a finite number of letters satisfying a followed by an infinite number of letters satisfying b (words that are "too long"). Recall that Formula 1 is equivalent to $[a\ \mathsf{U}\ [b\ \mathsf{U}\ c]]$. Formula 2, its weak version, is equivalent to the LTL formula $[a\ \mathsf{W}\ [b\ \mathsf{W}\ c]]$.

The problem of structural contradictions arises in the context of weak SERE formulas. Intuitively, a weak SERE formula r is related to its strong counterpart, $r!$, in the same way that the weak until operator W is related to its strong counterpart, U. For example, the formula $(p^* \cdot q)$ is equivalent to $[p\ \mathsf{W}\ q]$ and the formula $(p^* \cdot q)!$ is equivalent to $[p\ \mathsf{U}\ q]$. However, replacing q with a logical contradiction c gives a different result than replacing it with a structural contradiction s. While $(p^* \cdot c)$ is still equivalent to $[p\ \mathsf{W}\ c]$, the current semantics give that $(p^* \cdot s)$ is not equivalent to $[p\ \mathsf{W}\ s]$, but rather to *false*.

Structural contradictions also arise in the context of formulas using strong operators (e.g., $r!$ and U) on *truncated* words, words that are finite but not necessarily maximal. The problem of structural contradictions on such words is created by the PSL abort operator and the SVA disable iff operator. In [10] it was observed that methods developed for dealing with finite maximal words are not sufficient for dealing with truncated words. On truncated words, the user might want to reason about properties of the truncation as well as properties of the model. For instance, a user might want to specify that a simulation test goes on long enough to discharge all outstanding obligations, or on the other hand, that an obligation need not be met if it "is the fault of the test" (that is, if the test is too short). The former is useful for a test designed to continue until correct output can be confirmed, while the latter approach is useful for a test that "has no opinion" on the correct length of a test – for instance, a monitor checking for bus protocol errors.

The *truncated semantics* of LTL [10] gives three views of the validity of a formula on a truncated word. The views differ with respect to the truth value of a formula if the word was truncated before evaluation of the formula was complete. For example, consider the formula $\mathsf{F}\ p$ on a truncated word such that p does not hold for any letter, or the formula $\mathsf{G}\ q$ on a truncated word such that q holds at all letters. In both cases we cannot be sure whether or not the formula holds on the original, untruncated word.

A decision to return *true* when there is doubt is termed the *weak view* and a decision to return *false* when there is doubt is termed the *strong view*. Thus in the weak view the formula $\mathsf{F}\ p$ holds for any finite word, while $\mathsf{G}\ q$ holds only if q holds at every

[1] Weak SERE formulas are currently part of PSL but not of SVA. However, they are included in the Working Group-approved draft of the next version of SVA.

letter of the word. And in the strong view the formula $\mathsf{F}\ p$ holds only if p holds at some letter of the word, while the formula $\mathsf{G}\ q$ does not hold for any finite word. The *neutral view* demands the maximum that can be reasonably expected from a finite word. Under this approach, the formula $\mathsf{F}\ p$ holds only if p holds at some letter on the word, while the formula $\mathsf{G}\ q$ holds only if q holds at every letter on the word. This is exactly the traditional LTL semantics over finite words [19].

Following the truncated semantics, the formula F *false* holds in the weak view on a finite word, and the same sort of thing happens on strong SERE formulas in the current semantics of PSL and SVA. Thus the semantics can be understood as considering logical contradictions to be satisfiable in the weak view. The same does not hold, however, for structural contradictions, and the formula $\mathsf{F}\ (p \cap (p \cdot q))$ does not hold on any word in the weak view, and again, the same sort of thing happens on strong SERE formulas. We provide a solution to this anomaly in the form of a semantics that treats logical and structural contradictions in a consistent manner.

Our solution considers structural contradictions to be satisfiable in the weak view, in the same way that the current semantics of PSL and SVA consider logical contradictions satisfiable in that view. Why do we take this direction, rather than changing instead the treatment of logical contradictions? The reason is that changing the treatment of logical contradictions would have a huge impact on the complexity. In [4] it is shown that the *abort semantics* of an early version of PSL [1] has non-elementary complexity, which can be fixed by the *reset semantics*. These, in turn, were shown by [10] to be equivalent to the *truncated semantics*, which treat logical contradictions as satisfiable in the weak view. The difference in complexity is caused by the decision whether or not to treat logical contradictions as satisfiable in the weak view, thus changing the treatment of logical contradictions would return us to non-elementary complexity.

As we have seen, the idea of considering a logical contradiction to be satisfiable arose independently as a side effect of two very different motivations. While the motivation of the *reset semantics* presented in [4] was complexity, the equivalent *truncated semantics* [10] was motivated by the use of incomplete verification methods. The work presented in this paper shares the second motivation but not the first. Indeed, as we show, the complexity of model checking is not affected by the changes with respect to the existing semantics of PSL and SVA.

Finally, we emphasize that our main motivation is that if logical contradictions must be treated as satisfiable (for whatever reason) then we want logical and structural contradictions to be treated consistently, because intuitively there is no difference between asserting a logical contradiction and asserting a structural one.

2 Preliminaries

PSL and SVA are both temporal logics consisting of LTL [20] operators, formulas formed from semi-extended regular expressions (SEREs), clock and abort operators, and a lot of syntactic sugar.[2] It is of course not necessary to consider the syntactic sugar. Furthermore, clocks are orthogonal to the issues we examine in this paper (and as shown

[2] While the current version of SVA does not have LTL operators, they are included in the Working Group-approved draft of the next version of SVA.

Definition 1 (Semi-extended Regular Expressions (SEREs)). *If $b \in B$ is a propositional formula and r, r_1, and r_2 are SEREs, then the following are SEREs:*

- λ
- b
- $r_1 \cdot r_2$
- $r_1 \circ r_2$
- r^*
- $r_1 \cup r_2$
- $r_1 \cap r_2$

Definition 2 (PSC formulas). *If $b \in B$ is a propositional formula, φ and ψ are PSC formulas and r a SERE, then the following are PSC formulas:*

- $\neg\varphi$
- $\varphi \wedge \psi$
- $\mathsf{X!}\,\varphi$
- $[\varphi\ \mathsf{U}\,\psi]$
- $\varphi\ \mathsf{abort}\, b$
- $r!$
- r
- $r \mapsto \varphi$

Fig. 1. The syntax of PSC

by [12], can be written away by the rewrite rules of [17,18]), thus we use as our base logic the logic PSC (for PSL/SVA Core), consisting of all of the above-mentioned operators except for the clock operator. Since in the absence of the clock operator there is no difference between synchronous and asynchronous abort, we use here a single abort operator which we denote abort. We are left with LTL operators plus formulas formed from semi-extended regular expressions (SEREs) and the abort operator.

We define PSC formulas with respect to a non-empty set of atomic propositions P and a given set of propositional formulas B over P. Then SEREs and PSC formulas are defined as shown in Definitions 1 and 2 in Figure 1. Note that every propositional formula is a SERE, thus the weak and strong Booleans of [17] are included.

The weak next operator, X, is needed in order to deal with finite words [19] and is defined as syntactic sugaring as follows: $\mathsf{X}\ \varphi = \neg\mathsf{X!}\ \neg\varphi$. In this paper we also make use of $\mathsf{F}\ \varphi = [\mathit{true}\ \mathsf{U}\ \varphi]$ and $\mathsf{G}\ \varphi = \neg\mathsf{F}\ \neg\varphi$.

LTL is the subset of PSC consisting of degenerate SEREs of the form b as the base case, the Boolean connectives $\neg$ and $\wedge$, and the temporal operators $\mathsf{X!}$ and U.

2.1 Notation

Let $\Sigma = 2^P \cup \{\top, \bot\}$; thus a letter is either a subset of the set of atomic propositions P or one of the special letters $\top, \bot$. We will denote a letter from Σ by ℓ and an empty, finite, or infinite word from Σ by u, v or w. We denote the length of word w as $|w|$. An empty word $w = \epsilon$ has length 0, a finite word $w = (\ell_0\ell_1\ell_2\cdots\ell_n)$ has length $n + 1$, and an infinite word has length ∞. We denote the i^{th} letter of w by w^{i-1} (since counting of letters starts at zero). We denote by $w^{i..}$ the suffix of w starting at w^i. That is, $w^{i..} = (w^iw^{i+1}\cdots w^n)$ or $w^{i..} = (w^iw^{i+1}\cdots)$. We denote by $w^{i..j}$ the finite sequence of letters starting from w^i and ending in w^j. That is, $w^{i..j} = (w^iw^{i+1}\cdots w^j)$. We make use of an "overflow" and "underflow" for the indices of w. That is, $w^{j..} = \epsilon$ if $j \geq |w|$, and $w^{j..k} = \epsilon$ if $j \geq |w|$ or $k < j$.

The dual word of w, denoted $\overline{w}$, is the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We use *true* and *false* to denote $p \vee \neg p$ and $p \wedge \neg p$, respectively, for some atomic proposition p. We use φ and ψ to denote PSC formulas, p to denote an atomic proposition, and j and k to denote natural numbers.

For languages L_1 and L_2 we use $L_1 \cdot L_2$ to denote the set $\{w_1w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$ and $L_1 \circ L_2$ to denote the set $\{w_1\ell w_2 \mid w_1\ell \in L_1 \text{ and } \ell w_2 \in L_2\}$. For a language L

Definition 3 (The language of SERE r).

- $\mathcal{L}(\lambda) = \epsilon$
- $\mathcal{L}(b) = \{\ell \mid \ell \Vdash b\}$

- $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$
- $\mathcal{L}(r_1 \circ r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2)$
- $\mathcal{L}(r^*) = \mathcal{L}(r)^*$

- $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
- $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$

Fig. 2. The language of SERE r

we use L^0 to denote $\{\epsilon\}$, L^i to denote $L^{i-1} \cdot L$, L^* to denote $\bigcup_{i \geq 0} L^i$ and L^+ to denote $\bigcup_{i>0} L^i$. We use L^ω for the language composed of infinitely many concatenations of L with itself. For an infinite word w, we define that $wv = w$ for any word v.

We use $\langle a \rangle$ to denote a letter on which atomic proposition a and only atomic proposition a holds, $\langle ab \rangle$ to denote a letter on which atomic propositions a and b and only atomic propositions a and b hold, etc. Thus $\langle a \rangle\langle bc \rangle\langle d \rangle$ describes a finite word of three letters, such that a is the only atomic proposition that holds on the first letter, b and c are the only atomic propositions that hold on the second letter, and d is the only atomic proposition that holds on the third letter.

2.2 The Current Semantics of PSC

The current semantics of PSC in [17,18] is defined inductively, using as the base case the semantics of propositional formulas over letters in $\Sigma = 2^P \cup \{\top, \bot\}$. The semantics of propositional formulas is assumed to be given as a relation $\Vdash\ \subseteq \Sigma \times B$ relating letters in Σ with propositional formulas in B. If $(\ell, b) \in\ \Vdash$ we say that the letter ℓ satisfies the propositional formula b and denote it $\ell \Vdash b$. We assume that the two special letters $\top$ and $\bot$ behave as follows: for every propositional formula b, $\top \Vdash b$ and $\bot \nVdash b$. We further assume that on every other letter, $\ell \Vdash p$ iff $p \in \ell$. Finally, we assume that otherwise the relation $\Vdash$ behaves in the usual manner, and in particular that Boolean disjunction, conjunction and negation behave as usual.

The language of SERE r (the words on which r *holds tightly* [17], or is *tightly satisfied* [18]), denoted $\mathcal{L}(r)$, is defined formally as shown in Definition 3 in Figure 2.

Because of its use of the special letters $\top$ and $\bot$, we call the semantics of [17,18] the $\top$,$\bot$ approach to the semantics of PSC, and we use $w \models_{\top\bot} \varphi$ to denote that w models φ under the $\top$,$\bot$ approach. The semantics are given in Definition 4 in Figure 3. The semantics of the LTL operators are standard, except that we use the dual word when negating. The semantics of $r!$ are straightforward: $r!$ holds on a word w if there exists a finite prefix of w that is in $\mathcal{L}(r)$. For example, $(a \cdot b \cdot c)!$ holds on the word $u = \langle a \rangle\langle b \rangle\langle c \rangle\langle d \rangle$.

Intuitively, the semantics of r are that nothing should go wrong before reaching a final state in the automaton of r. In other words, that if we have not completed a word

Definition 4 (The current semantics of PSC).

- $w \models_{\top\bot} \neg\varphi \iff \overline{w} \not\models_{\top\bot} \varphi$
- $w \models_{\top\bot} \varphi \wedge \psi \iff w \models_{\top\bot} \varphi$ *and* $w \models_{\top\bot} \psi$
- $w \models_{\top\bot} \mathsf{X!}\,\varphi \iff |w| > 1$ *and* $w^{1..} \models_{\top\bot} \varphi$
- $w \models_{\top\bot} [\varphi\ \mathsf{U}\ \psi] \iff \exists k < |w|$ *such that* $w^{k..} \models_{\top\bot} \psi$ *and for every* $j < k,\ w^{j..} \models_{\top\bot} \varphi$

- $w \models_{\top\bot} r! \iff \exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$
- $w \models_{\top\bot} r \iff \forall j < |w|, w^{0..j}\top^\omega \models_{\top\bot} r!$
- $w \models_{\top\bot} r \mapsto \varphi \iff \forall j < |w|$ *s.t.* $\overline{w}^{0..j} \in \mathcal{L}(r), w^{j..} \models_{\top\bot} \varphi$

- $w \models_{\top\bot} \varphi\ \mathsf{abort}\, b \iff$ *either* $w \models_{\top\bot} \varphi$ *or* $\exists j < |w|$ *s.t.* $w^j \Vdash b$ *and* $w^{0..j-1}\top^\omega \models_{\top\bot} \varphi$

Fig. 3. The current semantics of PSC (the $\top$,$\bot$ approach)

in $\mathcal{L}(r)$, then at least things have not gone so wrong that appending some number of $\top$'s won't get us there. Thus the semantics are that for every prefix $w^{0..j}$ of w, $w^{0..j}\top^\omega$ should satisfy $r!$. For example, $(a \cdot b^* \cdot c)$ holds on the word $v_1 = \langle a\rangle\langle b\rangle\langle c\rangle\langle d\rangle$, and also on the "too short" word $v_2 = \langle a\rangle\langle b\rangle$ and on the "too long" word $v_3 = \langle a\rangle\langle b\rangle^\omega$.

The semantics of $r \mapsto \varphi$ (read "r *suffix implies* φ") are that whenever we see a prefix of w in $\mathcal{L}(r)$, we must have that φ holds on the continuation of w starting at the last letter of the prefix that "matched" r.[3] For example, the formula $(a \cdot b \cdot c) \mapsto \mathsf{G}\ d$ holds on the word $\langle a\rangle\langle b\rangle\langle cd\rangle\langle d\rangle^\omega$ because $\mathsf{G}\ d$ holds on the suffix of the word starting from the last letter that "matched" $(a \cdot b \cdot c)$. Note the use of the dual word $\overline{w}$. This is because implication entails a negation of the left operand. For example, the formula $(a \cdot b \cdot c) \mapsto \varphi$ does not hold on the word $w = \langle a\rangle\langle b\rangle\bot^\omega$ for any φ, because we have that $\overline{w} = \langle a\rangle\langle b\rangle\top^\omega$, and letting $j = 3$ we find the word $\langle a\rangle\langle b\rangle\top \in \mathcal{L}(a \cdot b \cdot c)$, but the continuation of the original word w starting from the third letter is $\bot^\omega$, on which φ surely does not hold (recall that nothing holds on $\bot$, not even *true*).

Finally, the semantics of the abort operator are that we truncate the word at the point where we see the abort condition, and pad the result with an infinite number of $\top$'s. Intuitively, this has the effect of weakening the strong operators on the truncated word. To see this, let the original word w be $w = uv$, and let the truncated word be u. Then a condition that must be fulfilled due to a strong operator can be fulfilled by one of the $\top$'s, if it is not fulfilled on u. For example, consider formula $\varphi = [p\ \mathsf{U}\ q]\ \mathsf{abort}\ b$ on word $w = \langle p\rangle\langle p\rangle\langle p\rangle\langle b\rangle$. Truncating w at b gives us $u = \langle p\rangle\langle p\rangle\langle p\rangle$. Then q holds on every $\top$ and in particular on the first one, so $[p\ \mathsf{U}\ q]$ holds on $u\top^\omega$.

3 Structural Contradictions

The $\top$,$\bot$ approach, used by both PSL [17] and SVA [18], is broken with respect to structural contradictions. We first define formally what we mean by a logical and a structural contradiction, then show the problem and our solution.

[3] In Dynamic Logic [13,15], $r \mapsto \varphi$ corresponds to $[r]\varphi$.

Definition 5 (Logical contradiction). *A logical contradiction is a propositional formula that is not satisfiable.*

Observation 1. *A logical contradiction can be transformed into a satisfiable (and valid) formula by substituting each proposition with either true or false.*

Definition 6 (Structural contradiction). *A structural contradiction is a* SERE *that is not satisfiable, and that remains unsatisfiable under every substitution of each proposition with either true or false.*

3.1 The Problem

Let $\varphi = (a \cdot b^{*} \cdot \mathit{false})$ and let $\varphi' = (a \cdot b^{*} \cdot (c \cap (c \cdot c)))$. It is easy to see that φ holds on $w = \langle a\rangle\langle b\rangle\langle b\rangle\langle b\rangle$ but that φ' does not. The reason for this is that any propositional formula holds on the letter $\top$, thus adding $\top$'s to the end of a finite word allows us to satisfy any logical contradiction in the future. However, there is no word, not even one consisting entirely of $\top$'s, on which a structural contradiction holds.

As suggested in [9], it seems that allowing the special letter $\top$ to match not only any propositional formula, but also any SERE of any length, might make the problem disappear by definition. Under the *flexible letter* approach, we modify the $\top$,$\bot$ approach by defining that $\mathcal{L}(\lambda) = \top^*$ and that $\mathcal{L}(b) = \{\ell \mid \ell \Vdash b\} \cdot \top^*$. We then get that $\varphi' = (a \cdot b^{*} \cdot (c \cap (c \cdot c)))$ holds on $w = \langle a\rangle\langle b\rangle\langle b\rangle\langle b\rangle$ as we want. To see this, note that for φ to hold on w we need that $w\top^\omega \models_{\top\!\bot} \varphi$. We have that $\top\top \in \mathcal{L}(c \cap (c \cdot c))$ because $\top\top \in \mathcal{L}(c)$ and also that $\top\top \in \mathcal{L}(c \cdot c)$. Thus we can let $j = 5$ and take the first six letters of $w\top^\omega$ to be in $\mathcal{L}(a \cdot b^{*} \cdot (c \cap (c \cdot c)))$.

The problem with this solution is that it has a harmful side effect – it changes the semantics of formulas that do not contain structural contradictions. Let $r_1 = (a \cdot b \cdot c)$, let $r_2 = (d \cdot e \cdot f)$, let $\varphi = (r_1 \circ r_2)$ and let $w = \langle a\rangle\langle b\rangle\langle c\rangle$. Then under the $\top$,$\bot$ approach, using the original semantics of $\mathcal{L}(r)$ as defined in Section 2.2, we have that φ does not hold on w. To see this, note that $|w| = 3$. Thus $w \models_{\top\!\bot} \varphi$ iff $\forall j < 3$ we have that $w^{0..j}\top^\omega \models_{\top\!\bot} (r_1 \circ r_2)!$. Taking $j = 2$ we get that $\langle a\rangle\langle b\rangle\langle c\rangle\top^\omega \not\models_{\top\!\bot} (r_1 \circ r_2)!$, because that requires one letter overlap between $(a \cdot b \cdot c)$ and $(d \cdot e \cdot f)$, but we do not have that d holds on the third letter. Under the flexible letter approach, using the modified definitions of $\mathcal{L}(\lambda)$ and $\mathcal{L}(p)$ of the previous paragraph, we can use more than three letters to "match" $(a \cdot b \cdot c)$. So take the first four letters of $\langle a\rangle\langle b\rangle\langle c\rangle\top^\omega$ to be in $\mathcal{L}(a \cdot b \cdot c)$, and then, starting from the last letter of those four (which is $\top$), take some number of letters to be in $\mathcal{L}(d \cdot e \cdot f)$. Since all letters after the first $\top$ are $\top$, we can take three $\top$'s for that. Then the overlap happens on a $\top$, and thus under the flexible letter approach we get that φ holds on w.

3.2 The Solution

Our proposed semantics for PSC is based on the *truncated semantics* of LTL [10]. The truncated semantics is defined over the natural alphabet 2^P and gives three views of the truth value of a formula, roughly corresponding to the truth value on $w\top^\omega$ (the weak view), w (the neutral view) and $w\bot^\omega$ (the strong view). Formally, the truncated

Definition 7 (Truncated semantics of LTL [10]).

holds weakly: *For w over 2^P such that $|w| \geq 0$,*

- $w \overset{-}{\models} b \iff |w| = 0$ *or* $w^0 \Vdash b$
- $w \overset{-}{\models} \neg\varphi \iff w \overset{+}{\not\models} \varphi$
- $w \overset{-}{\models} \varphi \wedge \psi \iff w \overset{-}{\models} \varphi$ *and* $w \overset{-}{\models} \psi$
- $w \overset{-}{\models} \mathsf{X!}\,\varphi \iff w^{1..} \overset{-}{\models} \varphi$
- $w \overset{-}{\models} [\varphi\ \mathsf{U}\ \psi] \iff \exists k$ *such that* $w^{k..} \overset{-}{\models} \psi$ *and for every* $j < k$, $w^{j..} \overset{-}{\models} \varphi$
- $w \overset{-}{\models} \varphi\ \mathsf{abort}\, b \iff$ *either* $w \overset{-}{\models} \varphi$ *or* $\exists j < |w|$ *s.t.* $w^j \Vdash b$ *and* $w^{0..j-1} \overset{-}{\models} \varphi$

holds neutrally: *For w over 2^P such that $|w| > 0$,*

- $w \models b \iff w^0 \Vdash b$
- $w \models \neg\varphi \iff w \not\models \varphi$
- $w \models \varphi \wedge \psi \iff w \models \varphi$ *and* $w \models \psi$
- $w \models \mathsf{X!}\,\varphi \iff |w| > 1$ *and* $w^{1..} \models \varphi$
- $w \models [\varphi\ \mathsf{U}\ \psi] \iff \exists k < |w|$ *such that* $w^{k..} \models \psi$ *and for every* $j < k$, $w^{j..} \models \varphi$
- $w \models \varphi\ \mathsf{abort}\, b \iff$ *either* $w \models \varphi$ *or* $\exists j < |w|$ *s.t.* $w^j \Vdash b$ *and* $w^{0..j-1} \overset{-}{\models} \varphi$

holds strongly: *For w over 2^P such that $|w| \geq 0$,*

- $w \overset{+}{\models} b \iff |w| > 0$ *and* $w^0 \Vdash b$
- $w \overset{+}{\models} \neg\varphi \iff w \overset{-}{\not\models} \varphi$
- $w \overset{+}{\models} \varphi \wedge \psi \iff w \overset{+}{\models} \varphi$ *and* $w \overset{+}{\models} \psi$
- $w \overset{+}{\models} \mathsf{X!}\,\varphi \iff w^{1..} \overset{+}{\models} \varphi$
- $w \overset{+}{\models} [\varphi\ \mathsf{U}\ \psi] \iff \exists k$ *such that* $w^{k..} \overset{+}{\models} \psi$ *and for every* $j < k$, $w^{j..} \overset{+}{\models} \varphi$
- $w \overset{+}{\models} \varphi\ \mathsf{abort}\, b \iff$ *either* $w \overset{+}{\models} \varphi$ *or* $\exists j < |w|$ *s.t.* $w^j \Vdash b$ *and* $w^{0..j-1} \overset{-}{\models} \varphi$

Fig. 4. The truncated semantics of LTL [10]

semantics of LTL is defined as shown in Definition 7 in Figure 4, where abort is the truncation operator, termed trunc_w by [10].

The neutral view is defined over non-empty words and corresponds to the traditional LTL semantics over finite words, while the weak and strong views are defined on empty words as well. The weak/strong views consider the empty word explicitly in the semantics of b (the weak view considers it sufficient that $|w| = 0$ while the strong view requires that w be non-empty), while the neutral view assumes that w is non-empty.

A third difference is that the weak and strong views make use of the definition of "overflow" for the indices of w. For example, in Definition 7, consider the semantics of $[\varphi\ \mathsf{U}\ \psi]$ under the weak and strong views. When we say "$\exists k$", k is not required to be less than $|w|$. In the weak view, this has the effect of allowing all eventualities to hold, because an overflow results in ϵ, b holds on ϵ, and the rest follows easily by induction.

Definition 8 (The weak language of SERE r).

- $\mathcal{L}_{weak}(\lambda) = \epsilon$
- $\mathcal{L}_{weak}(b) = \epsilon \cup \{\ell \mid \ell \Vdash b\}$

- $\mathcal{L}_{weak}(r_1 \cdot r_2) = \mathcal{L}_{weak}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{L}_{weak}(r_2))$
- $\mathcal{L}_{weak}(r_1 \circ r_2) = \mathcal{L}_{weak}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{L}_{weak}(r_2))$
- $\mathcal{L}_{weak}(r^*) = (\mathcal{L}(r)^* \cdot \mathcal{L}_{weak}(r)) \cup \mathcal{L}(r)^\omega$

- $\mathcal{L}_{weak}(r_1 \cup r_2) = \mathcal{L}_{weak}(r_1) \cup \mathcal{L}_{weak}(r_2)$
- $\mathcal{L}_{weak}(r_1 \cap r_2) = \mathcal{L}_{weak}(r_1) \cap \mathcal{L}_{weak}(r_2)$

Fig. 5. The weak language of SERE r

In the strong view, the overflow has the effect of not allowing any eventuality to hold, by similar reasoning.

The view is preserved in the inductive definition (e.g., $\wedge$ uses $\models^{-}$ in the weak view, $\models$ in the neutral view and $\models^{+}$ in the strong view) except that negation switches between the views, and aborting a formula always takes us to the weak view. Altogether, we get that the formula φ holds on a truncated word in the weak view if up to the end of the word, "nothing has yet gone wrong" with φ. It holds on a truncated word in the neutral view according to the standard LTL semantics on finite words. It holds on a truncated word in the strong view if everything that needs to happen in order to convince us that φ holds on the original, untruncated word, has already occurred.

For LTL formulas, the truncated semantics are equivalent to the $\top$,$\bot$ approach, as shown in [11].

Theorem 2 ([11]). *Let u be a word over 2^P, let v be a non-empty word over 2^P, and let φ be an LTL formula. Then, as shown in [11], the following equivalences hold:*

- $u \models^{-} \varphi \iff u\top^\omega \models_{\top\!\bot} \varphi$
- $v \models \varphi \iff v \models_{\top\!\bot} \varphi$
- $u \models^{+} \varphi \iff u\bot^\omega \models_{\top\!\bot} \varphi$

On infinite words, the weak, neutral and strong views coincide, as shown by [10]. Note that this follows trivially from the $\top$,$\bot$ approach, by the definition of concatenation to the right of an infinite word u.

We are now ready to extend the truncated semantics to all of PSC. As a first step, we can try to extend the definition of the *weak language* of a SERE [9] to the intersection and fusion operators. As defined in [9], the weak language of a SERE includes words that are in $\mathcal{L}(r)$, words that are "too short" (are finite prefixes of words in $\mathcal{L}(r)$), and also words that are "too long" (in which we get stuck in a starred sub-expression). The weak language of a SERE is defined in Definition 8 in Figure 5.

Using the weak language $\mathcal{L}_{weak}(r)$, we can define that

$$w \models r \iff \text{ either } \exists j < |w| \text{ s.t. } w^{0..j} \in \mathcal{L}(r) \text{ or } w \in \mathcal{L}_{weak}(r)$$

This appears to give us what we want, but there are two complications. The first is that we get that the formula $\varphi = (a \cdot b \cdot c) \circ (d \cdot e \cdot f)$ holds on the word $w = \langle a \rangle \langle b \rangle \langle c \rangle$. Thus we have changed the semantics with respect to formulas not containing structural contradictions, exactly the problem that we had with the flexible letter approach. To see this,

Definition 9. (The language of finite proper prefixes of SERE r) *The language of finite proper prefixes of* SERE *r, denoted $\mathcal{F}(r)$, is defined as follows:*

- $\mathcal{F}(\lambda) = \emptyset$
- $\mathcal{F}(b) \ = \epsilon$

- $\mathcal{F}(r_1 \cdot r_2) \ = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{F}(r_2))$
- $\mathcal{F}(r_1 \circ r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{F}(r_2))$
- $\mathcal{F}(r^*) \qquad = \mathcal{L}(r)^* \cdot \mathcal{F}(r)$

- $\mathcal{F}(r_1 \cap r_2) = \mathcal{F}(r_1) \cap \mathcal{F}(r_2)$
- $\mathcal{F}(r_1 \cup r_2) = \mathcal{F}(r_1) \cup \mathcal{F}(r_2)$

Definition 10. (The loop language of SERE **r)** *The loop language of* SERE *r, denoted $\mathcal{I}(r)$, is defined as follows:*

- $\mathcal{I}(\lambda) = \emptyset$
- $\mathcal{I}(b) \ = \emptyset$

- $\mathcal{I}(r_1 \cdot r_2) \ = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{I}(r_2))$
- $\mathcal{I}(r_1 \circ r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{I}(r_2))$
- $\mathcal{I}(r^*) \qquad = (\mathcal{L}(r)^* \cdot \mathcal{I}(r)) \cup \mathcal{L}(r)^\omega$

- $\mathcal{I}(r_1 \cap r_2) = \mathcal{I}(r_1) \cap \mathcal{I}(r_2)$
- $\mathcal{I}(r_1 \cup r_2) = \mathcal{I}(r_1) \cup \mathcal{I}(r_2)$

Fig. 6. The languages $\mathcal{F}(r)$ and $\mathcal{I}(r)$

let $r_1 = (a \cdot b \cdot c)$, let $r_2 = (d \cdot e \cdot f)$, and examine the semantics of $\mathcal{L}_{weak}(r_1 \circ r_2))$. Since we have already seen the third letter, we want to insist that d holds on it, as we do in the current semantics of PSC. But since $w \in \mathcal{L}_{weak}(r_1)$, we have not done so.

The second complication is that $\mathcal{L}_{weak}(r)$ does not distinguish between the case where formula r holds because the word is "too short" (is finite) and the case where it holds because the word is "too long" (is infinite). Thus it will be difficult for us to use $\mathcal{L}_{weak}(r)$ to extend the truncated semantics, in which it is required that the three views coincide on infinite, but not on finite words.

We address the first issue by defining a language that stops one letter short of words in $\mathcal{L}(r)$. In order to address the second issue, we split $\mathcal{L}_{weak}(r)$ into two languages. The language of proper prefixes of a SERE, denoted $\mathcal{F}(r)$, consists of *finite* proper prefixes of words in $\mathcal{L}(r)$, except that logical and structural contradictions are considered satisfiable. The loop language of a SERE, denoted $\mathcal{I}(r)$, extends the *loop*$(\cdot)$ of [16] to the intersection and fusion operators. It consists of *infinite* words in which we get "stuck forever" in a starred sub-expression of r. Formally, these languages are defined as in Definitions 9 and 10 in Figure 6.

Using $\mathcal{L}(r)$, $\mathcal{F}(r)$ and $\mathcal{I}(r)$, we extend the truncated semantics to all of PSC as shown in Definition 11 in Figure 7. Note that whenever we ask whether $w \in \mathcal{F}(r)$ we also ask if $w \in \{\epsilon\}$. This is because $\epsilon \notin \mathcal{F}(\lambda)$ (it recognizes only proper prefixes), however we want to preserve the property of [10] that any formula holds weakly on ϵ.

Examine now the semantics of $r!$ under the neutral view. The formula $r!$ holds neutrally on w if there exists a non-empty prefix of w that is in $\mathcal{L}(r)$. Since formula $r!$ is strong, its semantics under the strong view are identical to its semantics under the neutral view. Under the weak view, the semantics are weakened. The weakening includes the empty word and words in $\mathcal{F}(r)$, but not words in $\mathcal{I}(r)$. This is because the views should differ only on truncated words, which are finite, while words in the loop language are infinite.

The formula r holds neutrally on w if either there exists a non-empty prefix of w that is in $\mathcal{L}(r)$, or w is "too long" (is in $\mathcal{I}(r)$) or "too short" (is in $\mathcal{F}(r) \cup \{\epsilon\}$). Since formula r is weak, its semantics under the weak view are identical to its semantics under

Definition 11 (Truncated semantics of PSC). *The truncated semantics of* PSC *consists of the truncated semantics of* LTL *as shown in Definition 7 in Figure 4, extended to the* SERE*-based operators as follows:*

holds weakly: *For w over 2^P such that $|w| \geq 0$,*

- $w \overset{-}{\models} r! \iff$ *either* $\exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *or* $w \in \mathcal{F}(r) \cup \{\epsilon\}$
- $w \overset{-}{\models} r \iff$ *either* $\exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *or* $w \in \mathcal{I}(r)$ *or* $w \in \mathcal{F}(r) \cup \{\epsilon\}$
- $w \overset{-}{\models} r \mapsto \varphi \iff \forall j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *we have* $w^{j..} \overset{-}{\models} \varphi$

holds neutrally: *For w over 2^P such that $|w| > 0$,*

- $w \models r! \iff \exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$
- $w \models r \iff$ *either* $\exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *or* $w \in \mathcal{I}(r)$ *or* $w \in \mathcal{F}(r) \cup \{\epsilon\}$
- $w \models r \mapsto \varphi \iff \forall j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *we have* $w^{j..} \models \varphi$

holds strongly: *For w over 2^P such that $|w| \geq 0$,*

- $w \overset{+}{\models} r! \iff \exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$
- $w \overset{+}{\models} r \iff \exists j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *or* $w \in \mathcal{I}(r)$
- $w \overset{+}{\models} r \mapsto \varphi \iff w \notin \mathcal{F}(r) \cup \{\epsilon\}$ *and* $\forall j < |w|$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *we have* $w^{j..} \overset{+}{\models} \varphi$

Fig. 7. Truncated semantics of PSC

the neutral view. Under the strong view, the semantics are strengthened. Similarly to the weakening of $r!$ in the weak view, the strengthening affects only finite words.

The formula $r \mapsto \varphi$ holds neutrally on w if for every finite non-empty prefix of w in $\mathcal{L}(r)$, φ holds on the suffix of w starting from the last letter of the prefix. Formula $r \mapsto \varphi$ is weak, therefore its semantics under the weak view are similar to its semantics under the neutral view (the difference is in the strength used to check φ). Under the strong view, the semantics are strengthened. The strengthening involves eliminating ϵ and words that are in $\mathcal{F}(r)$. To understand this, notice that on such words there might exist an extension to a word in $\mathcal{L}(r)$ on the original, untruncated word (considering logical and structural contradictions to be satisfiable) and we cannot be sure whether φ holds on the original, untruncated word starting from that point.

Before examining the characterisitics of the truncated semantics of PSC, let's do a quick sanity check. Previously we saw that on a finite word $w = \langle a \rangle \langle b \rangle \langle b \rangle \langle b \rangle$, formulas $\varphi = (a \cdot b^* \cdot \mathit{false})$ and $\varphi' = (a \cdot b^* \cdot (c \cap (c \cdot c)))$ behave differently under the $\top,\bot$ approach. Under the extension to the truncated semantics, we get that both of them hold on w. Thus it seems that in some sense the truncated semantics of PSC is treating logical and structural contradictions in a consistent manner. In the next section we formalize this and other characteristics of our solution.

4 Characteristics of the Truncated Semantics of PSC

In this section we show that our extension of the truncated semantics to all of PSC treats logical and structural contradictions in a consistent manner. We show that it preserves important properties of the original truncated semantics, and that for formulas without structural contradictions, it preserves the semantics of the $\top$,$\bot$ approach. Finally, we show that the complexity of model checking is the same under the truncated semantics of PSC as under the $\top$,$\bot$ approach. We start with a simple proposition.[4]

Proposition 3 (*false* vs. *true* $\cap$ (*true* $\cdot$ *true*)). *Let φ be a* PSC *formula containing false and let φ' be the formula obtained by replacing every occurrence of false with the structural contradiction true $\cap$ (true $\cdot$ true). Let w be a finite or infinite word over 2^P. Then the truth values of φ and φ' on w agree under the truncated semantics of* PSC.

We would like to extend Proposition 3 to cover any logical vs. structural contradiction, and not just *false* vs. *true* $\cap$ (*true* $\cdot$ *true*). However, things are not so simple. While every logical contradiction is equivalent to every other logical contradiction, not all structural contradictions are created equal. For example, let $r_1 =$ *true* $\cap$ (*true* $\cdot$ *true*) and let $r_2 =$ (*true* $\cdot$ *true*) $\cap$ (*true* $\cdot$ *true* $\cdot$ *true*). Then r_2 holds on words of length 1, while r_1 does not. To understand this, note that a structural contradiction contains an element of temporality that is not present in a logical contradiction. So we should compare a structural contradiction to a temporal logic formula that also contains such an element. For example, while r_1 is equivalent to *false*, r_2 is equivalent to *true* $\cdot$ (*true*$\cap$(*true* $\cdot$ *true*)), which is equivalent to X *false*.

We define the *order* of a formula to be the length of the longest word on which it holds weakly. Thus *false* and $p \cap (p \cdot q)$ are of order 0 and X *false* and $(p \cdot q) \cap (p \cdot q \cdot r)$ are of order 1, and we expect that a generalization of Proposition 3 would compare only formulas of the same order. Note first that two formulas of order n are not necessarily equivalent. For example, X *false* holds on any word of length 1, whereas $(p \cdot q) \cap (p \cdot q \cdot r)$ holds only on a subset of such words – those where p holds on the first letter.

Let $\mathsf{X}^0\ \varphi$ denote φ, let X^n denote n repetitions of the X operator and let r^n denote n concatenations of r with itself. Then Proposition 4 below generalizes Proposition 3.

Proposition 4 (X^{n-1} *false* vs. *true*^{n} $\cap$ (*true*^{n} $\cdot$ *true*$^+$)). *Let $n \geq 1$, let φ be a* PSC *formula containing sub-formula $\psi = \mathsf{X}^{n-1}$ false and let φ' be the formula obtained by replacing every occurrence of ψ with the structural contradiction $\psi' =$ true^n $\cap$ (true^n $\cdot$ true$^+$). Let w be a finite or infinite word over 2^P. Then the truth values of φ and φ' on w agree under the truncated semantics of* PSC.

The observant reader may have noticed that the structural contradiction $\lambda \cap$ *true* is not addressed by Proposition 4. It has order 0, and it might be expected that *true* $\cap$ (*true* $\cdot$ *true*) $=$ *true* $\cdot$ ($\lambda \cap$ *true*) have order 1 (it actually has order 0). This apparent anomaly is resolved if we consider a clocked semantics [12]. Under a clocked semantics, $\mathsf{X}^0\ \varphi$ is not equivalent to φ, but is the *clock alignment operator*, and we have that $\lambda \cap$ *true* is equivalent to *false* while *true* $\cap$ (*true* $\cdot$ *true*) is equivalent to X^0 *false*. The details are beyond the scope of this paper – for a thorough discussion of clock alignment, see [14]. Finally, note that all structural contradictions of the form $\lambda \circ r$ are of order 0, and are equivalent to *false*.

[4] All proofs appear in the full version of this paper, available at http://www.cs.huji.ac.il/danafi/publications.

Proposition 4 deals with structural contradictions of a specific structure. What about others, for example $\lambda \cap \mathit{true}$, $(\mathit{true}^{13} \cap \mathit{true}^{7}) \cup (\mathit{true}^{9} \cap \mathit{true}^{4})$, $(a^6 \cup a^4) \cap a^7$, or $(a \cdot b) \cap (a \cdot b \cdot c \cdot d \cdot e)$? Are they satisfiable in the weak view? The answer is yes, all structural contradictions are satisfied on the empty word under the weak view (and none is satisfied by the empty word in the strong view), as stated in the following lemma.

Lemma 5. *Let φ be a formula in* PSC. *Then both $\epsilon \models^{-} \varphi$ and $\epsilon \not\models^{+} \varphi$.*

Lemma 5 fulfills our intuition that on the empty word clearly nothing has yet gone wrong (thus it should accept weakly any formula, including a structural contradiction) and conversely the empty word provides evidence for nothing (thus it should accept no word strongly, including a structural contradiction). Lemma 5 is important because it shows that a fundamental property of the original truncated semantics, one that was broken in the $\top$,$\bot$ approach, is preserved by the truncated semantics of PSC. Below we show that other important properties of the original truncated semantics are preserved by our extension to all of PSC. Theorem 6 states that the strong view is stronger than the neutral, which is in turn stronger than the weak.

Theorem 6 (Strength relation theorem). *Let w be a non-empty word over 2^P, and φ be a formula in* PSC. *Then*

- $w \models^{+} \varphi \Longrightarrow w \models \varphi$
- $w \models \varphi \Longrightarrow w \models^{-} \varphi$

Corollary 7 states that all three views are equivalent over infinite words.

Corollary 7. *If w is infinite, then $w \models^{-} \varphi$ iff $w \models \varphi$ iff $w \models^{+} \varphi$.*

Finally, Theorem 8 states that if a formula holds weakly on a word w then it holds weakly on all prefixes of w, and if a formula holds strongly on a word w then it holds strongly on all extensions of w. We say that u is a prefix of v, denoted $u \preceq v$, if there exists a word u' such that $uu' = v$. We say that w is an extension of v, denoted $w \succeq v$, if there exists a word v' such that $w = vv'$.

Theorem 8 (Prefix/extension theorem). *Let v be a word over 2^P, and φ be a formula in* PSC. *Then*

- $v \models^{-} \varphi \Longleftrightarrow \forall u \preceq v,\ u \models^{-} \varphi$
- $v \models^{+} \varphi \Longleftrightarrow \forall w \succeq v,\ w \models^{+} \varphi$

Previously we have seen that even without structural contradictions, the flexible letter approach and the solution using $\mathcal{L}_{weak}(r)$ differ from the semantics of the $\top$,$\bot$ approach. The theorem below states that without structural contradictions, the $\top$,$\bot$ approach and our extension to the truncated semantics agree.

Theorem 9. *Let u be a word over 2^P, v be a non-empty word over 2^P, and φ be a* PSC *formula that does not contain a* SERE *that is a structural contradiction. Then*

- $u \models^{-} \varphi \Longleftrightarrow u\top^{\omega} \models_{\top\!\bot} \varphi$
- $v \models \varphi \Longleftrightarrow v \models_{\top\!\bot} \varphi$
- $u \models^{+} \varphi \Longleftrightarrow u\bot^{\omega} \models_{\top\!\bot} \varphi$

Theorem 9 shows that the special letters $\top$ and $\bot$ are tools that are useful in stating the $\top$,$\bot$ approach, but are not necessary in stating the semantics in the absence of structural contradictions. Besides the fact that they don't give us the semantics that we want, a major disadvantage of the $\top$,$\bot$ approach is that it uses the special letters $\top$ and $\bot$ to encrypt the semantics, rather than stating directly what they are, as we do in the extension to the truncated semantics.

Intuitively, the weak view can be obtained by considering all states of a Büchi automaton to be accepting, similarly to the *safety component* of [3] and to the *looping acceptance condition* of [21]. However, we have to be careful. The particular automaton that results from applying the looping acceptance condition depends on the form of the automaton that we start with, and not just on its original language. For instance, consider the Büchi automaton consisting of a single (not final) state and no transitions. Obviously, its language is empty. Now add a single transition on the letter $\langle a \rangle$. Since the single state is not final, we have not changed the language of the automaton, which remains empty. However, applying the looping acceptance condition to the original and the modified automata will result in different languages. The following theorem shows that there are automata of the same size as those implementing the current semantics of PSL/SVA, for which a manipulation of this sort works.

Theorem 10. *Let r be a* SERE *and let φ be a* PSC *formula.*

- *There exist non-deterministic finite automata (*NFW*) $\mathcal{N}_{\mathcal{L}}(r)$, $\mathcal{N}_{\mathcal{F}}(r)$ and a Büchi automaton (*NBW*) $\mathcal{B}_I(r)$ each with $O(2^{|r|})$ states that accept $\mathcal{L}(r)$, $\mathcal{F}(r)$ and $\mathcal{I}(r)$, respectively. Moreover, these automata agree on all components but the set of accepting states.*
- *There exists an alternating Büchi automaton (*ABW*) B_φ with $O(2^{|\varphi|})$ states that accepts exactly the set of words w such that $w \models \varphi$.*
- *The satisfiability and model checking problems for formulas in* PSC *are EXPSPACE-complete.*

Theorem 10 shows that the proposed semantics is not harder than the current semantics, and thus that complexity is not a motivation for preferring one over the other, or for considering structural contradictions to be satisfiable. This is in contrast to logical contradictions, for which, as shown by [4], complexity is an important consideration.

5 Conclusion and Future Work

We have presented a semantics that treats logical and structural contradictions in a consistent manner. Finding a solution was complicated by the need to support not only intersection, but also fusion and the three views (which are needed by abort). We defined three separate languages for SEREs. The language $\mathcal{L}(r)$ is the traditional language of a semi-extended regular expression. The language of proper prefixes of a SERE, denoted $\mathcal{F}(r)$, consists of *finite* proper prefixes of words in $\mathcal{L}(r)$, except that logical and structural contradictions are considered satisfiable. The loop language of a SERE, denoted $\mathcal{I}(r)$, consists of *infinite* words in which we get "stuck forever" in a starred subexpression of r. A bonus is that our semantics are defined on the natural alphabet 2^P rather than the inflated alphabet $2^P \cup \{\top, \bot\}$, currently used by both PSL and SVA. The complexity of model checking PSC is not affected by the changes we propose.

Previously [9], we used a topological characterization to prove that without the intersection and fusion operators, the relation between $r!$ and r is the same as that between strong and weak until. Because that characterization is based on the $\top,\bot$ approach, it breaks down when we admit the intersection and fusion operators. Future work is to find a topological characterization for the solution that we have presented here.

References

1. Accellera Property Specification Language Reference Manual Version 1.0 (January 2003)
2. Accellera Property Specification Language Reference Manual Version 1.1 (June 2004)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987)
4. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Resets vs. aborts in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
5. Ben-David, S., Bloem, R., Fisman, D., Griesmayer, A., Pill, I., Ruah, S.: Automata construction algorithms optimized for PSL (Deliverable 3.2/4). Technical report, Prosyd (2005)
6. Bustan, D., Fisman, D., Havlicek, J.: Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science (May 2005)
7. Bustan, D., Havlicek, J.: Some complexity results for SystemVerilog assertions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 205–218. Springer, Heidelberg (2006)
8. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
9. Eisner, C., Fisman, D., Havlicek, J.: A topological characterization of weakness. In: Proc. PODC 2005, pp. 1–8. ACM Press, New York (2005)
10. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
11. Eisner, C., Fisman, D., Havlicek, J., Mårtensson, J.: The $\top,\bot$ approach to truncated semantics. Technical Report 2006.01, Accellera (May 2006)
12. Eisner, C., Fisman, D., Havlicek, J., McIsaac, A., Van Campenhout, D.: The definition of a temporal clock operator. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 857–870. Springer, Heidelberg (2003)
13. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. 18(2), 194–211 (1979)
14. Fisman, D.: On the characterization of until as a fixed point under clocked semantics. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 19–33. Springer, Heidelberg (2008)
15. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
16. Harel, D., Sherman, R.: Looping vs. repeating in dynamic logic. Information and Control 55, 175–192 (1982)
17. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850$^{\text{TM}}$-2005, Annex B (2005)
18. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800$^{\text{TM}}$-2005, Annex E (2005)
19. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
20. Pnueli, A.: The temporal logic of concurrent programs. Theoretical Computer Science 13, 45–60 (1981)
21. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

Synthesizing Test Models from Test Cases

Antti Jääskeläinen, Antti Kervinen, Mika Katara, Antti Valmari, and Heikki Virtanen

Tampere University of Technology
Department of Software Systems
P.O.Box 553, FI-33101 Tampere, Finland
{antti.m.jaaskelainen,firstname.lastname}@tut.fi

Abstract. In this paper we describe a methodology for synthesizing test models from test cases. The context of our approach is model-based graphical user interface (GUI) testing of smartphone applications. To facilitate the deployment of model-based testing practices, existing assets in test automation should be utilized. While companies are interested in the benefits of new approaches, they may have already invested heavily in conventional test suites. The approach presented in this paper enables using such suites for creating complex test models that should have better defect detection capability. The synthesis is illustrated with examples from two small case studies conducted using real test cases from industry. Our approach is semi-automatic requiring user interaction. We also outline planned tool support to enable efficient synthesis process.

1 Introduction

Model-based software testing [1] has several obvious advantages over conventional test suite testing where test cases are crafted manually. For instance, on-line tests generated from state machines can reach significantly higher coverage in testing non-deterministic systems under test (SUTs) than linear and static test suites. Moreover, maintenance of large test suites is more difficult when changes occur in the SUT. Frequent changes are common especially in graphical user interface (GUI) testing that is typically used to check the functionality of the SUT from the perspective of the end users before a release is made.

The problems with conventional test automation approaches have resulted in many bad experiences, and manual testing is still widely considered as the primary quality assurance method at the system and acceptance level testing of GUI-intensive software [2]. While unit and integration level test automation can significantly improve code quality and enable efficient refactoring, system level test automation entails much more challenges. This is due to the *domain-specific nature* of system level testing; at the unit and integration levels all SUTs seem more or less similar, depending on the programming language used; the same white-box testing and static analysis techniques work across different domains. At the system level, however, the context comes into play: testing a banking system can be quite different from testing a set-top box.

The deployment of model-based system testing has been hampered in many contexts in spite of its many benefits [3,4]. In our earlier work, we have developed a domain-specific solution to the GUI testing of S60 [5] smartphone applications that should be easier to deploy than more generic methodologies [6,7]. The approach consists of a domain-specific modeling language based on LSTSs (Labeled State Transition Systems) augmented with S60 specific restrictions, a model-library containing test models for the basic smartphone applications such as calendar, contacts, camera, and messaging, and tools for on-line test generation. In on-line testing, the idea is to generate tests while they are executed, thus testing can be seen as a game between the test automation system and the SUT [8].

In the course of developing our approach we have identified another problem in deployment: companies may have invested huge sums of money to craft test suites and thus can be unwilling to invest to the development of test models replacing the former way of working. Thus, in order to facilitate the deployment of our approach, we have developed a semi-automatic method for synthesizing test models from test cases. This enables utilizing the existing assets when moving from test suite testing to model-based one. The method is domain-specific to enable a higher level of automation in the synthesis and promote the usefulness of the resulting models. However, a similar method could presumably be developed for some other domain, using similar principles.

In this paper we describe the method and the case studies we have conducted. In addition, since model synthesis is quite different from the traditional way of creating models, and we compare the synthesized model to a one crafted by hand using a top-down approach [9]. A tool support for the synthesis is also outlined; its implementation will be future work. The remainder the paper is structured as follows: Section 2 describes the context of our contributions, i.e., model-based GUI testing of mobile applications. Then, we move on to present our approach for model synthesis in Section 3. Sections 4 and 5 present the case studies and discuss the results and the future work.

2 Model-Based GUI Testing of Mobile Software

Action words and keywords [10,11] are commonly used concepts in software test automation, especially in GUI testing. The basic idea is to separate different concerns: *what* are the important actions to be tested and *how* they are implemented. Action words are high level descriptions of functionality; in the smartphone context there can be different action words for opening the messaging application, taking a photo with the camera, or adding a new contact, for instance. Keywords, on the other hand, specify the exact sequence of events that are needed to implement the functionality described by an action word. In S60 GUI, for instance, there can be multiple ways of opening a messaging application (short cut, menu, some other application). Each of the different ways can be encoded as a separate sequence of key strokes that accomplish the action. Furthermore, to receive input from the SUT, some keywords can be dedicated to verifying that a given text string is found on the display, for instance.

The main benefit of action words and keywords is in enabling non-technical testers to design action word level tests without deep knowledge of the underlying keyword implementations. Moreover, they ease the tedious maintenance tasks often hindering the use of GUI test automation; in many cases minor GUI changes can be restricted to the keyword level. Action words and keywords can be used in conventional approaches so that the keywords are implemented as a library of functions, one function for each keyword. Action words are then specified using spread sheets, for instance, that list the sequences of keywords needed to implement the corresponding action word. Finally, test cases can be encoded as sequences of action words using spread sheets as in the previous step.

However, linear and static tests are limited in their ability to find new defects. Thus, the true power of the action words and keywords is realized when combined with automatic test generation based on behavioral models. For this purpose, we have chosen to use Labeled State Transition Systems (LSTSs) [12] for test modeling. LSTS is an extension of the more common Labeled Transition System (LTS) formalism where labels have been added to states as well as transitions. Action words and keywords are used as transition labels in the models. The formal definition for LSTS is as follows:

Definition 1 (LSTS). *A* labeled state transition system*, abbreviated LSTS, is defined as a sextuple* $(S, \Sigma, \Delta, \hat{s}, \Pi, val)$ *where* S *is the set of* states*,* Σ *is the set of* actions *(transition labels),* $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions*,* $\hat{s} \in S$ *is the* initial state*,* Π *is the set of* attributes *(state labels) and* $val : S \longrightarrow 2^{\Pi}$ *is the* attribute evaluation function*, whose value* $val(s)$ *is the set of attributes in effect in state* s*.*

Notation of internal transitions makes no sense in test modeling, because our behavioral models have to be strictly deterministic for test generation. Our definition differs from the original one in that respect.

Actions can be divided into three categories according to how they deal with the SUT: *input*, *output* and *setup actions*. Input actions correspond to user input, and output actions get information from the SUT. Setup actions affect the SUT just as input actions, but in ways not accessible to an ordinary user. Setup actions might, for example, directly create or remove files in memory or alter internal settings. Action words often combine aspects of more than one category, whereas keywords usually fall neatly into one or another.

To enable modular and compositional test modeling, *parallel composition* is used for combining test model components. The parallel composition of LSTSs [12] is based on a rule set explicitly defining which actions are executed synchronously. An action of the composed LSTS can be executed only if the corresponding actions can be executed in each component LSTS, or if the component LSTS is indifferent to its execution. The following definition is slightly modified in two respects; internal transitions are not needed and handling of state propositions is made more straightforward:

Definition 2 (Parallel composition $\|_R$). $\|_R (L_1, \ldots, L_n)$ *is the* parallel composition *of LSTSs* $L_1, \ldots, L_n$*,* $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i, \Pi_i, val_i)$*, according to* rules R*, with* $\forall i, j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$*. Let* Σ_R *be a set of resulting actions and* $\surd$ *a "pass" symbol such that* $\forall i; 1 \leq i \leq n : \surd \notin \Sigma_i$*. The rule set* $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$*. Now* $\|_R (L_1, \ldots, L_n) = repa((S, \Sigma, \Delta, \hat{s}, \Pi, val))$*, where*

– $S = S_1 \times \cdots \times S_n$
– $\Sigma = \Sigma_R$
– $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *if and only if there is* $(a_1, \ldots, a_n, a) \in R$ *such that for every* i $(1 \leq i \leq n)$ *either*
  • $(s_i, a_i, s'_i) \in \Delta_i$ *or*

    - $a_i = \surd$ *and* $s_i = s_i'$
- $\hat{s} = (\hat{s}_1, \dots, \hat{s}_n)$
- $\Pi = \Pi_1 \cup \dots \cup \Pi_n$
- $val((s_1, \dots, s_n)) = val_1(s_1) \cup \dots \cup val_n(s_n)$
- *repa is function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$.

Parallel composition offers tools for implementing rudimentary variables, which the basic LSTS formalism lacks. A variable can be created as a single component model, whose states correspond to different values. The actions in such a *variable model* are synchronized to those of the other component models so that different values allow different actions. These synchronized actions can be used to test the value of the variable or to change it. The idea of using compositional test modeling and separate variable components is motivated by existing tools, that proof the concept [13].

To hide the complexity inherent in test models and test generation algorithms, and so to facilitate the deployment of our model-based testing methodology, we have introduced a web based testing service [7]. The idea is that test service users can order tests using a simple web interface specifying the desired coverage requirements. The coverage requirements are then used for driving on-line test generation based on an extensive model library containing test models for basic S60 applications [9].

We believe that such a service can greatly ease the adoption of model-based testing in smartphone application testing. However, companies have existing assets in conventional test suites, and they might prefer to utilize them when migrating from traditional test suite based automation to a model-based one. This led us to research an approach for synthesizing test models from test cases.

3 Synthesis of Test Models

The synthesis process we have developed allows the creation of a single test model from a number of test cases. The cases must be strictly linear to begin with; they should also be specific in detail. The resulting model will have the same level of abstraction (action word/keyword) as the original cases. Test cases which verify the state of the SUT often may be easier to handle, but the process is designed to also work with few or no verifications.

The process has five distinct phases. In the first phase the relevant actions are listed and parameterized. The second phase consists of creating variables to hold some of the state information of the SUT. The third phase takes care of the initialization sequence of the SUT. In the fourth phase recurring states within the test cases are marked and labeled. Finally, the fifth phase sees the test cases merged together with the variables and the initialization to form a new test model.

Although the phases are presented consecutively, their order is not fixed. Only the merging phase is dependent on the others and must therefore be performed last. The others may be performed in any order, and it may even be a good idea to consider them side by side. Throughout the process description we will present a running example, starting with the three imaginary action word level test cases in Figure 1. In the first the phone

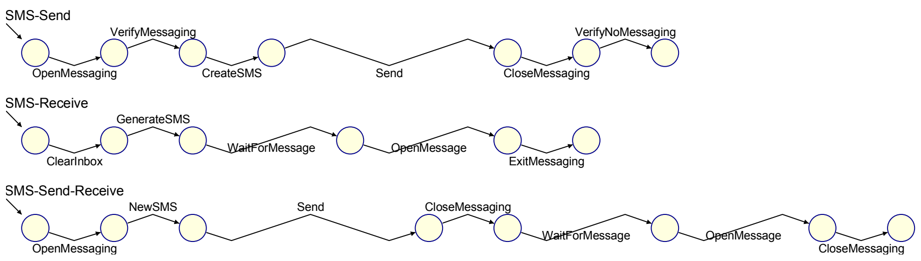


Fig. 1. The three initial example test cases

sends an SMS to itself, in the second it receives and opens an automatically generated SMS, and in the third it first sends an SMS and then receives it. Note that the actions *CreateSMS* and *GenerateSMS* perform the same task, as do the actions *CloseMessaging* and *ExitMessaging*. They are used to demonstrate the effects of different actions sequences corresponding to the same functionality.

3.1 Action Definition

The first thing to do is to list all the actions used within the source test cases. Possible parameters should not be included. Once listed, each action is assigned two values: weight and idempotence status.

An action's weight represents its situational specificity. An action with a high weight is one whose execution with a certain parameter is likely to lead the SUT into the same state every time. This may be either because the action is only executable in very few states or because it resets parts of the SUT. An action with a low weight, on the other hand, is one which can be executed in many different situations and whose effects depend on the current situation. Weights are used in the merging of test cases. If identical action sequences taken from different test cases or different parts of the same test case have a high combined weight, it is likely that the sequences are related to the same functionality of the SUT. If this is the case, the two test cases may be merged at the points after the sequences, giving them two different ways to proceed from that point. The comparison is made with sequences instead of single actions because a long series of actions is likely to be far more situationally specific than any of its actions individually.

Actions may be marked as idempotent. The execution of an idempotent action leaves the SUT in the state it had before the execution. Most idempotent actions are used to get information out of the SUT. An idempotent action can be discarded from a test case without breaking it, although the testing value of the case may drop.

Finding the right weights is not an exact process. Action words should generally be given high weights, whereas keywords' weights vary case by case. In our running example all actions are action words. This means they have a high situational specificity, and we can give all of them maximal weights. VerifyNoMessaging and VerifyMessaging are idempotent, the rest are not.

Following are some examples with keywords: A keyword for resetting the SUT has a very high weight, since by default it always leaves the SUT in the same state. It is clearly not idempotent. A keyword which verifies that a given text is visible on the screen is idempotent and has a relatively high weight, since the same text does not very often occur in different situations. A keyword indicating that nothing should be done for a period of time has minimal weight, since waiting is always possible. It is not idempotent, because it is generally used in situations where the state of the SUT is expected to change during the wait.

3.2 Variable Definition and Integration

Embedding a part of the state of the SUT into variables is an important part of the synthesizing process. Without separate variables, the states of the test cases may contain so much information that they can never be merged together. The first, most difficult task is to identify the variables to be created. As a general rule, those properties of the SUT which are independent of the current screen of the SUT yet affect execution should be moved to variables. Having too few variables reduces the number of potential merge points and thereby limits the functionality of the final model. Too many variables mean more work in creating them and may increase the size of the final model, but should not reduce its quality.

After the variables have been determined, each is given a number of possible values. The number of values should be kept as small as possible, because they can cause exponential growth in the final model. Once the values have been chosen, each may be given one or more setup actions as *assignment actions*. In the final model, the execution of the assignment action will automatically set the variable into the designated value. A single action may act as an assignment action for multiple values, as long as they do not belong to the same variable. Finally, for each variable one of its values may be chosen as the initial value. The initial value should either have an assignment action or be otherwise guaranteed when testing begins. A variable may be left uninitialized, but then no action based on it can be taken until it has been given a value during a test run, and the size of the resulting model is also somewhat increased.

Once the variable definitions are ready, variable models are created for them. For this purpose we have made a simple Python script which reads in the variable definitions in CSV (Comma Separated Values) format and automatically produces an LSTS for each variable. The script also creates a *variable initialization model* which can set the variables to specific values before a test run by using the assignment actions.

The ready variables must be integrated into the test cases. This is performed by adding preconditions and postconditions to the actions in the test cases. Preconditions specify the values of the variables necessary for the successful execution of the action. Postconditions, conversely, define the changes of values caused by the execution of the action. Assignment actions do not require explicit postconditions, but are synchronized directly into appropriate variables. For optimal result, pre- and postconditions should be placed right around the relevant action, not around a whole action sequence.

In our running example, we create a single variable to record whether there is a message on its way to the phone, so that we will be free to merge the test cases at the main screen, regardless of whether messages have been sent or not. We use GenerateSMS

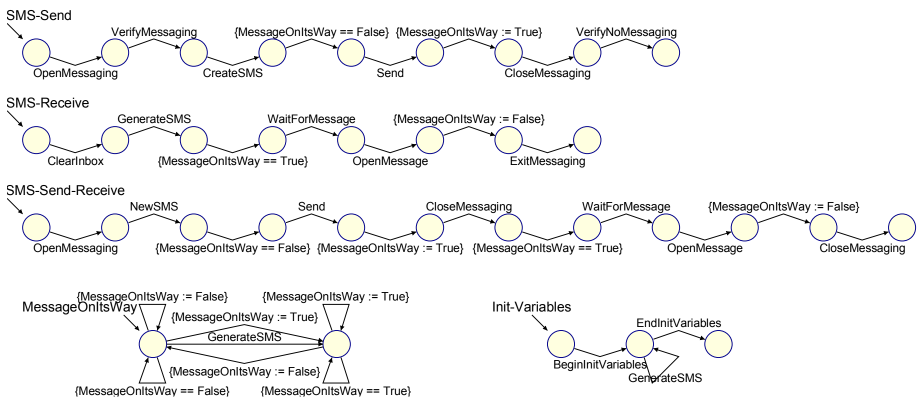

SMS-Send
OpenMessaging
VerifyMessaging
CreateSMS
{MessageOnItsWay == False}
Send
{MessageOnItsWay := True}
CloseMessaging
VerifyNoMessaging
SMS-Receive
ClearInbox
GenerateSMS
{MessageOnItsWay == True}
WaitForMessage
OpenMessage
{MessageOnItsWay := False}
ExitMessaging
SMS-Send-Receive
OpenMessaging
NewSMS
{MessageOnItsWay == False}
Send
{MessageOnItsWay := True}
CloseMessaging
{MessageOnItsWay == True}
WaitForMessage
OpenMessage
{MessageOnItsWay := False}
CloseMessaging
MessageOnItsWay
{MessageOnItsWay := False}
{MessageOnItsWay := True}
{MessageOnItsWay := True}
GenerateSMS
{MessageOnItsWay := False}
{MessageOnItsWay == False}
{MessageOnItsWay == True}
Init-Variables
BeginInitVariables
EndInitVariables
GenerateSMS

Fig. 2. The example test cases with pre- and postconditions added

as an assignment action for the value True and pick False as the initial value, which should be safe for a new test run. Figure 2 shows the variable model and the variable initialization model, and above them the test cases with pre- and postconditions marked with braces.

3.3 Initialization Sequence Definition

In order to automatically set the SUT into its initial state before a test run, an initialization sequence is defined. The sequence contains those setup actions which should always be executed before a test run. They could, for example, reset the SUT, disable features that might interfere with testing, and create suitable data. Variable initialization should not be included here. As a rule, all setup actions should be within the initialization sequence or act as an assignment action for a variable. If a setup action belongs to neither group, more variables might be needed.

The rest of the initialization phase could be performed automatically with the information from the earlier phases, although we do not currently have tools for it. The initialization sequence is made into a *general initialization model*. All non-idempotent setup actions are removed from the beginnings of the test cases (by now they are all in the general initialization model or the variable initialization model), and synchronization is added to connect them into the initialization models.

The changes made into the test cases in the example are very minor, as Figure 3 shows. The only setup action is ClearInbox, which has been moved into a model of its own.

3.4 State Label Definition and Assignment

The existence of the variables allows the test cases to be merged with relative freedom, but there is no guarantee that suitable merging points can be automatically identified. For this purpose *state labels* are added into the test cases. The important states of the

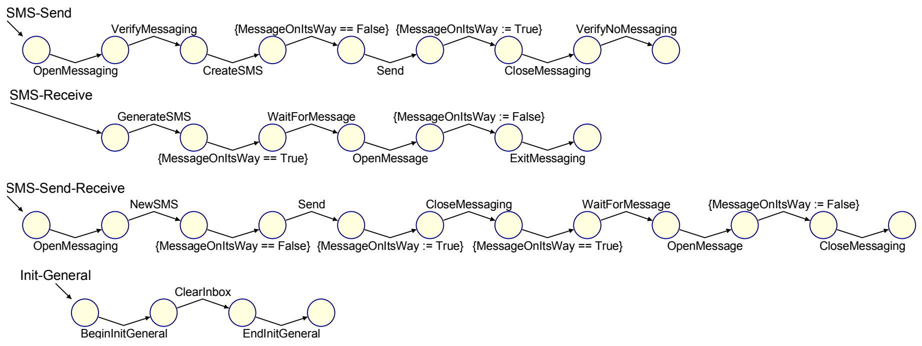


Fig. 3. The example test cases with setup actions separated

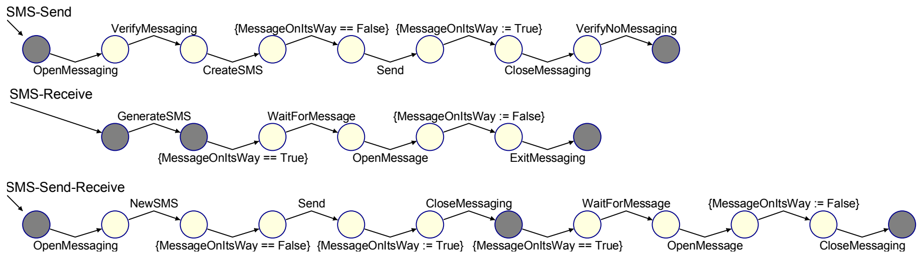


Fig. 4. The example test cases with filled states marking the main screen

SUT are identified and a name is given to each. Especially important are the starting and ending states of the test cases (ideally the same state); the basic states of other major SUT screens visited during the test cases are also good choices. Properties included in variables should be ignored.

Once the important states have been selected, state labels with suitable parameters are placed into test cases at every point in which the SUT is in a chosen state. The state labels can be handled as LSTS attributes; alternatively they can be interpreted as idempotent actions with maximal weights. Either way, merges will always be attempted at their points of execution. They can be easily removed from the final model so that they do not interfere with its execution.

In our example, we decide that the only noteworthy state is the main screen of the phone and label it, as shown in Figure 4. The states in question have been filled.

3.5 Merging of the Component Models

Now that the test cases have been prepared we can perform the actual merging. This is done with the merger program, which looks for identical sequences of sufficient weight within the test cases and suggests merging their destination states. The program may also offer false suggestions, i.e. merges that would result in an erroneous model. Because of this, the legality of each merge must be manually checked to ensure the validity

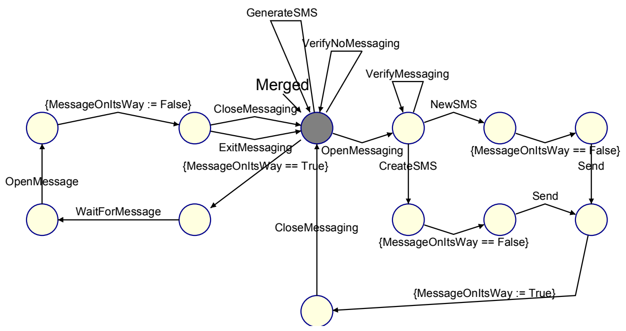


Fig. 5. The model merged from example test cases

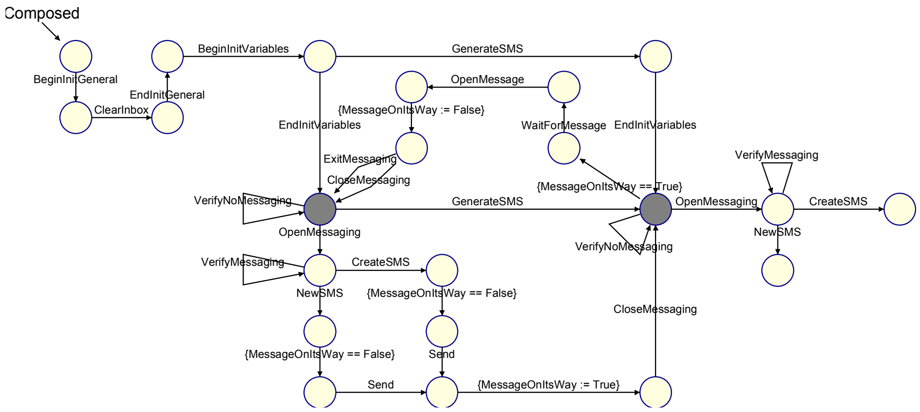


Fig. 6. The example model after parallel composition

of the model. The merging results in a *control model* which contains the functionality of the original test cases, but without the information encoded into variables. The merged model, variable models and initialization models are then passed through parallel composition, which creates an executable test model.

While the test model obtained this way is usable, it may pose difficulties for test generation. That is because the model is likely to contain many paths leading to deadlocks, i.e. states with no outgoing transitions, resulting either from a denied precondition or the end of a test case that could not be merged anywhere. The model may be cleaned by removing all the dead paths, but this is not always a good idea. If the test cases could be looped back into themselves and deadlocks occur only or mostly in places where a precondition fails, the clean-up procedure should be safe to perform. Conversely, if many test cases ended in unique states and caused deadlocks at the end, the clean-up could remove relevant functionality. In this case the model may be better left as-is, and the test generation algorithm must take care not to guide the execution toward a deadlock prematurely.

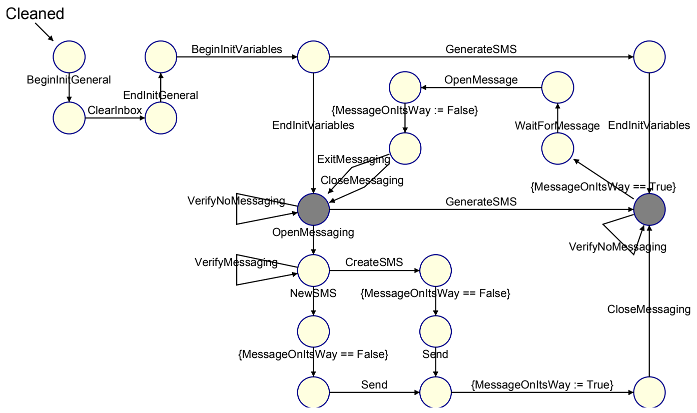


Fig. 7. The final, cleaned model

Figure 3.5 shows the model obtained from our example test cases. Merges have been performed at matching actions and state labels. Adding the initialization models and the variable model in parallel composition results in the usable model depicted in Figure 6. It would seem that in this case the cleaned model (Figure 7) would be more useful, since the dead end on the right likely serves no practical purpose; it depicts a situation where a new message is created with one already on its way, causing the preconditions to block its sending.

4 Case Studies

We have performed two small scale case studies to test our synthesizing methodology. The original sequences were linear keyword level test cases picked from a much larger set of test cases for S60 applications developed by one of our industrial partners. The first case study used seven test cases for the Phonebook application. The second one had nine for the Messaging application, concentrating on short and multimedia messages (SMS and MMS). Both case studies used the same set of 30 keywords. The Phonebook test cases had 193 actions altogether, the Messaging test cases 363. Three of the test cases for the Messaging case study can be seen in Figure 8, with some changes made for readability and to adapt them for a single phone.

Both case studies used the same set of keywords, which we were already familiar with from our earlier work. Giving keywords their weights was therefore easily done, though the values were somewhat arbitrary; we had yet to perform enough experiments to find the best values. The Phonebook case proved to require seven variables, six to hold information about existing contacts and groups and one for incoming messages. The Messaging case required six variables, two for the existence of messages and reports and the rest for various settings. The first case labeled the idle state and the contacts and groups screens, the latter labeled the idle state and the screens for SMS and MMS writing.

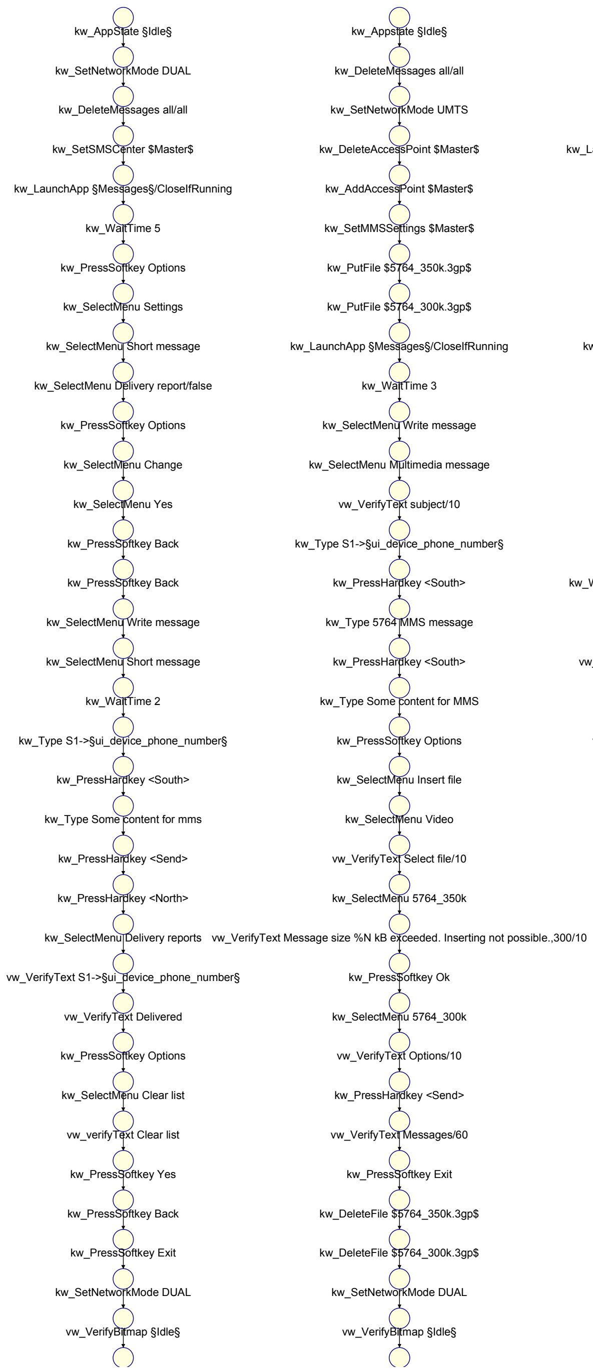


Fig. 8. Three of the nine Messaging test cases

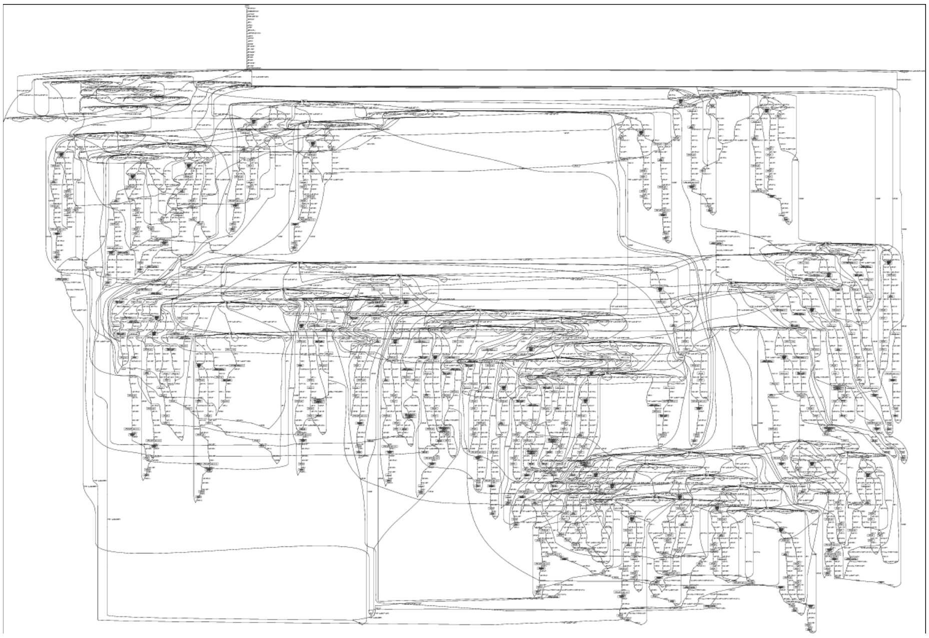


Fig. 9. The final Messaging model

After the merge, the Phonebook model had 126 and the Messaging model 192 states. Parallel composition and cleanup brought state counts to 12523 and 2327, respectively. The Phonebook case shows the potentially exponential growth caused by variables. This happened because the variables controlled relatively small portions of the model and had little to do with each other. Conversely, the variables in the Messaging case were interconnected to some degree, and affected control to a much greater extent; for example, many individual test cases specified certain settings before sending a message. As a result, large portions of the control model were reachable only with certain variable values. Figure 9 shows an overview of the final Messaging model, illustrating its scope and complexity. Although the models are too large for human understanding, their size is not a problem for our automated test generation tools.

The quality of the final models appeared to be comparable to the test models in our test model library [9] created by hand from scratch, although not quite equal to them. The synthesized models contained less functionality, but this was a result of the original choice of test cases, not a failing of the method itself. A notable difference was the higher granularity of the synthesized models: often actions which could be performed separately in hand-made models were forcibly chained together in synthesized ones. However, this tendency did not seem to reach truly detrimental levels, and the number of possible action sequences was still magnitudes higher than in the original linear test cases. The final difference between the synthesized models and our old models was that keyword level test cases naturally became a single keyword level model, not a combination of keyword and action word level models as in our model library. The action word level might be added using the bottom-up modeling technique presented in [6]. Presumably action word level test cases could be combined into an action word level model and the action words then refined as in original test cases, though we have yet to attempt that.

In both case studies, most of the effort during the synthesizing process went into variable definition and integration. In the Phonebook case, this was mostly manual work: the variables were simple, but referenced often. With Messaging the situation was different. There much time was spent in deciding what exactly should be modeled into variables, and how exactly would they be integrated into the test cases. Placing the pre- and postconditions also took considerable time, mostly because the complexity of the variables demanded great care in integrating them into control. We found merging to be relatively easy, but it might pose more difficulties to someone not used to test modeling. It definitely requires some understanding of the implemented variables, which implies that the whole process might be best performed by a single person.

Both of the case studies were performed by a single person and each required less than a day to complete. It seems quite reasonable to us that with good tools a person familiar with the process could synthesize a model of considerably greater size within a single day. That would be notably faster than creating a comparable test model from scratch, and would not require a similar expertise in modeling. Fortunately the most time-consuming phase, variable definition and integration, seems likely to scale reasonably well with the number of test cases (probably linear effort or less). The least scalable phase by far is merging of the component models (potentially quadratic or even exponential effort), which at least might be fully automatable.

5 Discussion

In this paper we have described an approach for synthesizing test models from test cases. In addition, we presented the results of two small case studies where the approach was applied for creating test models from existing test cases in the domain of S60 GUI testing. The synthesis is semi-automatic and thus requires user interaction to achieve useful results. A tool supporting this interaction was also sketched.

Our approach is domain-specific in the sense that the set of keywords and the corresponding weight values must be decided based on the domain knowledge. In our case studies this was easy because the same person who had built our model library conducted the experiments. However, the other phases of the synthesis process should be applicable also in other contexts.

There exists a large body of knowledge about the synthesis process. While most, if not all, of the existing approaches have been originally developed for design, analysis and code generation purposes, they may be useful for test model synthesis also. Amyot and Eberlein have compared twenty-six solutions for constructing design models from scenarios [14]. Moreover, Liang, Dingel, and Diskin have developed comparison criteria for comparing different algorithms and applied the criteria to compare twenty-one different approaches [15]. However, it seems that domain knowledge can improve the synthesis; we first experimented with a more generic approach [16], but decided to develop our own to better fit the needs of our context. An extensive study would be needed to analyze the other existing approaches for their applicability to test model creation, but this lies outside the scope of this paper.

Some of the currently manual phases in our synthesizing process might be automated, most notably initialization and parts of modeling of variables. The action weights and state labels must be set manually. The defining and integration of variables also requires user input, but actual variable models can be created automatically. It might also be possible to automate merging totally, not just finding the potential merge points. In the two case studies, potential merge points occurring at state labels were always mergeable; this seems likely to be a general rule, as long as the labels have been placed well. The merge points based on action sequences varied, some being mergeable and others not. However, in these cases the sequence merges did nothing that could not have been replicated with well-placed state labels. Based on these observations, it might be possible to automate the merging to always merge at labels and disregard sequences altogether, but more testing is required before implementing such changes.

Although the most work-intensive part of the process, the creation and integration of variables, cannot be truly automated, it could be substantially eased by proper tools. These should offer both an easy way to define variables, preferably hiding the models altogether, and a simple method for setting pre- and postconditions. Some algorithm for suggesting potential variables would be a highly useful feature, but difficult to design.

In the future, in addition to developing tool support, there is also the need to conduct wider case studies and to compare the test coverage that can be achieved with handcrafted versus synthesized test models in actual on-line test generation.

Acknowledgements

This paper reports results of research funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, and Plenware, as well as the Academy of Finland (grant number 121012). For details, see http://practise.cs.tut.fi/project.php?project=tema.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007)
2. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing: A Context-Driven Approach. Wiley, Chichester (2001)
3. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 118–127 (2003)
4. Hartman, A.: AGEDIS project final report (2004) (cited June 2008), http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF
5. S60. (Cited June 2008), http://www.s60.com
6. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: Proceedings of TAIC PART – Testing: Academic & Industrial Conference, Windsor, UK, pp. 81–89. IEEE Computer Society, Los Alamitos (2006)

7. Jääskeläinen, A., Katara, M., Kervinen, A., Heiskanen, H., Maunumaa, M., Pääkkönen, T.: Model-based testing service on the web. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 38–53. Springer, Heidelberg (2008)
8. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, USA, pp. 55–64. ACM, New York (2004)
9. Jääskeläinen, A., Kervinen, A., Katara, M.: Creating a test model library for GUI testing of smartphone applications. In: Proceedings of the 8th International Conference on Quality Software (QSIQ 2008), pp. 276–282. IEEE Computer Society, Los Alamitos (2008)
10. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison Wesley, Reading (1999)
11. Buwalda, H.: Action figures. STQE Magazine, 42–47 (March/April 2003)
12. Hansen, H., Virtanen, H., Valmari, A.: Merging state-based and action-based verification. In: Proceedings of ACSD 2003, the Third International Conference on Application of Concurrency to System Design, pp. 150–156. IEEE, Los Alamitos (2003)
13. Virtanen, H., Hansen, H., Valmari, A., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 153–157. Springer, Heidelberg (2004)
14. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommunication Systems 24, 61–94 (2003)
15. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2006), pp. 5–12 (2006)
16. Mäkinen, E., Systä, T.: MAS - an interactive synthesizer to support behavioral modeling in UML. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 15–24. IEEE Computer Society Press, Los Alamitos (2001)

D-TSR: Parallelizing SMT-Based BMC Using Tunnels over a Distributed Framework

Malay K. Ganai and Weihong Li

NEC Labs America, Princeton, NJ, USA

Abstract. We present a tool D-TSR for parallelizing SMT-based BMC over a distributed environment targeted for checking safety properties in low-level embedded (sequential) software. We use a *tunneling and slicing-based* reduction (TSR) approach to decompose disjunctively a BMC instance (at a given depth) into simpler and independent subproblems. We exploit such a decomposition to cut down communication cost and idle time of CPUs during synchronization while solving BMC instances. Our approach scales almost linearly with number of CPUs, as demonstrated in our experimental results.

1 Introduction

Bounded Model Checking (BMC) provides a complete design coverage with respect to a correctness property for a bounded depth. In spite of using richer expressive theories such as SMT (Satisfiability Modulo Theory in a decidable subset of first order logic) to obtain a compact representation, each BMC instance grows bigger in size and harder to solve with successive unrolling of the design. One possible solution is to use a distributed environment to solve each BMC instance. Due to communication overhead and inherent synchronization in such an environment, there are many challenges in achieving linear scalability, or even close to it. Further, due to uneven (and often unpredictable) work loads and unreliable worker machines, the problem becomes all the more challenging.

Several techniques for distributed BMC have been proposed previously [1,2,3]; however, they do not address the scalability requirements adequately. In a distributed BMC approach [1], each BMC instance is partitioned structurally so that each processor gets an exclusive number of consecutive BMC time frames. The distributed problem is then solved by a distributed SAT, managed by a central server. Though this method overcomes the memory limitation of a single processor, and employs fine grain parallelization of a SAT solver, it incurs significant communication overhead during exchanges of lemmas and propagation of values across partitions. In [2], each BMC instance is solved independently on a separate client. As each BMC instance is not partitioned, there is a significant slow down as unrolling depth increases. In [3], an initial partition is generated using partial assignments for initial states and properties being verified. Each partition is sent to a client, which solves the BMC problem for depths 1 to some bound N using the partial state assignments. Note, each client solves the entire BMC problem, albeit for different initial states. One can use parallel SAT-solvers such as [4,5] to solve each BMC instance. In general, parallelizing a DPLL-based SAT solver incurs large communication and synchronization costs. Our presented approach is orthogonal to the above-mentioned approaches.

1.1 Tool Overview

We present a tool D-TSR for parallelizing SMT-based BMC over a distributed environment, where we partition disjunctively a BMC instance (at a particular depth) using a *Tunneling and Slicing-based Reduction* (TSR) approach into smaller and independent subproblems, based on *tunnels*, i.e., a set of control paths [6]. Each sub-problem is simplified further and solved independently. Our distribution framework comprises a single controller (master) and several workers (clients).

- We address communication overhead by creating tunnels *statically* and *deterministically*. Such partitioning is a light-weight operation, and is performed by each client at every BMC unrolled depth. When the master is required to assign a sub-problem, it simply notifies a chosen client with a corresponding tunnel id.
- We address uneven (unpredictable) load balancing using *relaxed synchronization criteria*. Since each sub-problem is independent, the master need not wait for the clients that are slow to respond due to slower CPUs or due to harder sub-problems assigned, at the time of synchronization. The master dynamically adjusts a pool of available clients; the slower clients are removed from the pool at the time of synchronization (after each BMC depth), and added back when they respond. This reduces the idle time for (faster) clients.

We focus on verifying low-level embedded programs under the assumptions of a finite recursion and bounded data, in a distributed environment. We formulate common design errors such as array bounds violations, null pointer de-referencing, and user-provided assertions as reachability properties, and solve them using BMC.

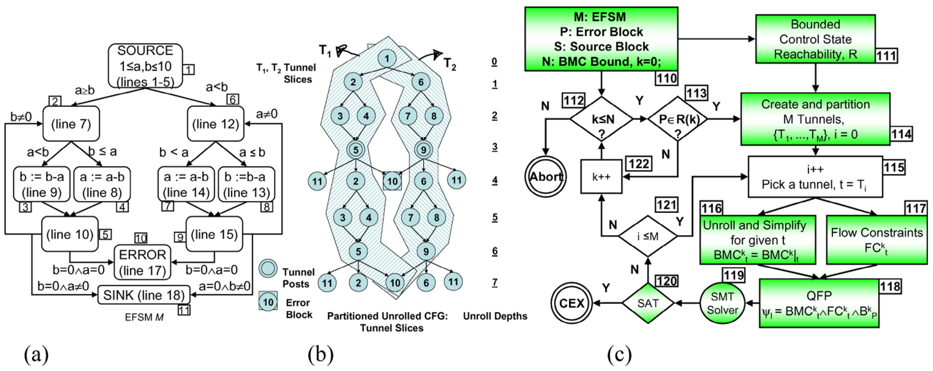


Fig. 1. (a) EFSM M, (b) Tunnels T_1 and T_2 of unrolled CFG, (c) TSR-based BMC

2 TSR-Based BMC

Consider an EFSM (Extended Finite State Machine) M shown in Figure 1(a). We use a tool F-Soft [7] to obtain EFSM from a low level C program. In the following, we use a TSR-based BMC (shown in Figure 1(c)), to illustrate the reachability of an ERROR control state, $P = 10$ from a SOURCE control state $S = 1$.

We obtain an unrolled CFG (Control Flow Graph) by simply unwinding the CFG up to a depth $k = 7$ as shown in Figure 1(b) (block 111, Figure 1(c)). Each program (concrete) path in a BMC instance up to a depth k, is an instance of an (abstract) control path in the unrolled CFG. As the unrolled depth increases, number of paths (and control paths) and their lengths also increase, thereby, making each successive BMC instance increasingly harder to solve. As shown in Figure 1(b), the number of control paths to reach ERROR block 10 increases from 4 to 8, as k increases from 4 to 7, respectively. We use $R(k)$ to denote a set of control states statically reachable at a depth k from SOURCE state. *Example:* $R(4) = \{2, 10, 11, 6\}$.

We decompose a BMC instance BMC^k, say at $k = 7$, into smaller disjunctive subproblems by partitioning control paths as follows (block 114, Figure 1(c)). We pick a partition depth $p = 3$. At this depth p, control states 5 and 9 are statically reachable. All control paths from control state 1 (SOURCE) to control state 10 (ERROR) at depth 7, pass through a control state either 5 or 9 at a depth $p = 3$. We refer these control states as *tunnel-posts*. We partition these tunnel-posts disjunctively into sets, $\{5\}$ and $\{9\}$. From the control state(s) in each partitioned set, i.e., $\{5\}$ (or $\{9\}$), ERROR block at a depth 7, and SOURCE block at a depth 1, we perform forward and backward slicing on the unrolled CFG to obtain a disjoint set of control paths, i.e., T_1 (or T_2) as shown in Figure 1(b). We refer to them as *tunnels*. Note, all control paths in *tunnels* T_1 and T_2 pass through the partitioned *tunnel-posts* $\{5\}$ and $\{9\}$, respectively at a partition depth $p = 3$. For each partitioned tunnel T_1 or T_2, we obtain a BMC subproblem $BMC^k_{T_1}$ or $BMC^k_{T_2}$ by constraining the BMC problem with the respective tunnels. For a given tunnel t, we first simplify a BMC subproblem BMC^k_t by (a) *slicing* away the irrelevant (i.e., not in t) paths, (b) *reducing* the data path expressions in t, and (c) *adding* flow constraints (FC^k_t) for the relevant paths [8] (blocks $116, 117$, Figure 1(c)). Note, such a BMC subproblem has fewer paths, and is potentially easier to solve than the original BMC instance. We formulate each reduced subproblem with the property constraint B^k_P, denoted as $BMC^k|_t \wedge FC^k_t \wedge B^k_P$, as a quantifier-free formula (QFP) and solve it separately and independently using a SMT-solver. Note, the satisfiability of a BMC subproblem implies satisfiability of the BMC instance. Further, due to the independency of subproblems, we effectively obtain an efficient decomposition for (potential) parallelization.

3 D-TSR: Distributed TSR-Based BMC

The tasks (partially shaded boxes in Figure 1(c)) are distributed between master and clients as shown in Figures 2(a)-(b), respectively, where master-clients are connected in a star topology. Note, in a star-topology, a client can communicate with the master only. In the following, we use $< tasks : id >$ to refer tasks in block with id in Figure 1(c).

3.1 Tasks of the Master

The tasks of the master are shown in Figure 2(a). Let A_c denote a set of available clients, W_c denote a set of current busy clients, A'_c denote a set of clients that are available but not included in A_c, and W'_c denote a set of previous busy clients but not included in W'_c, respectively. Initially (block 210), the master sets A_c to all available clients, and sets W_c, A'_c and W'_c to $\emptyset$.

In blocks 210 and 211, it performs tasks $< tasks : 110, 111 >$ and $< tasks : 112, 113, 122 >$, respectively. In block 212, it creates M partitions for a BMC at depth

k ($< tasks : 114 >$). In block 213, it notifies all the clients in the set A_c to carry out `TSR_INIT:k` (described in Section 3.2). In blocks 214−217, it distributes M partitions among A_c clients on a first available basis. For a first available client c it assigns a partition i by sending `TSR_SOLVE:i` message. Note, it sends only the partition id, not the entire partition information. It updates A_c and W_c sets, accordingly.

If there exists an unassigned partition, but no available client (i.e., all are busy solving an assigned task), it waits for some client to respond (block 219) with `TSR_STATUS`. If a client c responds (block 220), it checks if $c \in W_c$. If not, it updates the sets A'_c and W'_c that track out-of-order messages, i.e., messages corresponding to `TSR_SOLVE` of previous BMC instance (block 221). (Such out-of-order messages could arise when some partition is found satisfiable, and master did not wait for a reply from a client solving another partition of same BMC instance.) Otherwise, the sets A_c and W_c are updated (block 222). It checks if the status received is SAT (block 223). If not, it assigns an unsolved partition to a next available client. If the status is SAT (i.e., BMC instance is satisfiable), it sends `TSR_ABORT` to all clients in the set W_c (block 224). It then waits for clients to respond (block 218) using a simple *relaxed synchronization* criteria. As per that, *instead of waiting for all clients that are yet to respond, it only waits for some number of clients so that the total number of available clients, i.e., $|A_c|$, is above some threshold.*

After waiting, it updates A_c and W'_c, sends `TSR_QUIT` to clients in A_c, and then, continues to solve a new BMC instance at depth $k + 1$.

3.2 Tasks of a Client

The tasks of a client are shown in Figure 2(b). Initially, a client performs operations $< tasks : 110, 111 >$ and then waits for a message from the master.

If the received message is `TSR_INIT:k`, it carries out static TSR partitioning $< tasks : 114 >$ of BMC instance at depth k. As the partitioning of TSR is deterministic, the partitioning by both master and a client produces identical results, and therefore, each partition can be identified unambiguously by them. If a received message is `TSR_SOLVE:i`, the client unrolls $< tasks : 116 >$, simplifies and adds learning

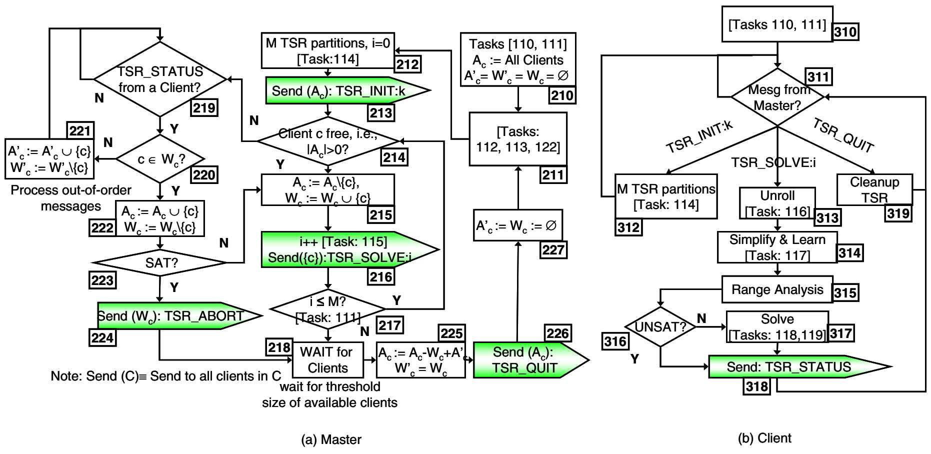


Fig. 2. Tasks of (a) the Master (controller), and (b) a Client (worker)

constraints $< tasks : 117 >$ for the partition i. It performs light-weight symbolic range analysis to check if the BMC instance is UNSAT.

If the result is unsatisfiable, it sends `TSR_STATUS:UNSAT` message to the master; otherwise, it invokes a SMT solver to check if the partition i is satisfiable $< tasks : 118, 119 >$. In block 317, it also checks periodically if there is a message `TSR_ABORT` (not shown). If such a message exists, it simply aborts and replies master with `TSR_STATUS:ABORT` message; otherwise, it sends `TSR_STATUS:SAT` or `TSR_STATUS:UNSAT` depending on the satisfiability outcome.

If a received message is `TSR_QUIT`, it cleans up the memory used after last `TSR_INIT` message.

4 Experimentation

We implemented our tool D-TSR using standard communication library LAM/MPI [9] available publicly. We used several workstations each with 4Gb RAM and Intel multi-cores (2 to 8 cores) cpu, each with speed in the range of 1.8 to 2 Ghz. In our experiment, we scheduled at most one client per cpu. We used a SMT-based BMC framework [8, 10, 6], using `yices-1.0` [11] as the SMT solver.

We present result for `f1` which is a restart module of *wu-ftpd* with *array bound violation* checks.

For scalability results, we compare our SMT-based BMC performance implemented in our D-TSR using different number of client-CPUs, i.e., 1, 2, 4, 8, and 16. We denote each configuration as `para#` where (#) denotes the number of client-CPUs used. Note, `para1` is equivalent to a sequential BMC with TSR partitioning with (small) communication overhead. We also compare against an implementation of SMT-based BMC (`mono`) without using TSR. We allocated one hour for each BMC run.

We present our results, i.e., BMC depth vs Cummulative Time (in sec)) in Figure 3. The vertical line in the chart shows the time taken (in sec) by each client-configuration for a specific unroll depth 466. Observe that `para1` gives substantial speedup over `mono`, indicating that the TSR partitioning improves the performance of BMC even without parallelization. With distribution of tasks, we obtain an almost linear-speedup. We believe that such approach can easily scale to a large number of CPUs.

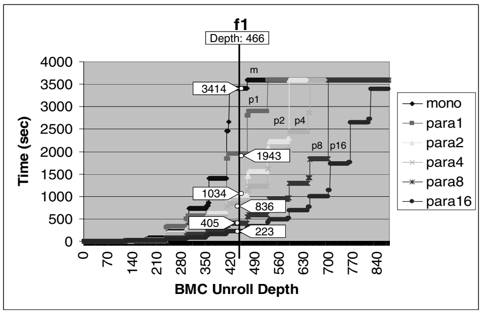


Fig. 3. Depth v/s Time chart

References

1. Ganai, M.K., Gupta, A., Yang, Z., Ashar, P.: Efficient distributed SAT and SAT-based distributed bounded model checking. Journal on STTT 8(4-5), 387–396 (2006)
2. Abraham, E., Schubert, T., Becker, B., Franle, M., Herde, C.: Parallel sat solving in bounded model checking. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 301–315. Springer, Heidelberg (2007)
3. Barros, H., Campos, S., Song, M., Zarate, L.: Exploring clause symmetry in a distributed bounded model checking algorithm. In: ECBS (2007)
4. Jurkowiak, B., Li, C.M., Utard, G.: A parallelization scheme based on work stealing for a class of SAT solvers. Journal of Automatic Reasoning 34(1), 73–101 (2005)
5. Blochinger, W., Sinz, C., Kuchlin, W.: Parallel propositional satifiability checking with distributed dynamic learning. Parallel Computing 29(7), 969–994 (2003)
6. Ganai, M.K., Gupta, A.: Tunneling and Slicing: Towards Scalable BMC. In: Proc. of DAC (2008)
7. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
8. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: Proc. of ICCAD (2006)
9. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium, pp. 379–386 (1994)
10. Ganai, M.K., Gupta, A.: Completeness in SMT-based BMC for software programs. In: Proc. of DATE (2008)
11. SRI. Yices: An SMT solver, http://fm.csl.sri.com/yices

Progress in Automated Software Defect Prediction

Thomas J. Ostrand and Elaine J. Weyuker

AT&T Labs - Research, Florham Park, NJ 07932, U.S.A.
{ostrand,weyuker}@research.att.com

Abstract. We have designed and implemented a tool that predicts files most likely to have defects in a future release of a large software system. The tool builds a regression model based on the version and defect history of the system, and produces a list of the next release's most probable fault-prone files, sorted in decreasing order of the number of predicted defects. Testers can use this information to decide where to focus resources, and to help determine how much effort to allocate to various parts of the system. Developers can use the tool's output to help decide whether files should be rewritten rather than patched. A prototype version of the tool has been integrated with AT&T's internal software change management system, providing seamless access to the system's version and defect information, and giving users a simple interface to the tool's output.

Keywords: software fault prediction, negative binomial model, automated tool.

1 Research Overview

The goal of this research has been the development of fully automatable models to allow developers and testers to identify the files of a large industrial software system that are most likely to contain the largest numbers of faults in the future. Based on these models, we have built a tool capable of making predictions with minimal human intervention, expertise, or time. The tool is designed to automatically extract data both from the repository that is normally used by the change management system and from the software code itself. The data is analyzed and used to construct appropriate statistical models, which can predict where faults are most likely to reside.

Testing practitioners can use this information to prioritize their testing efforts, and developers can use it to determine which fault-prone files should be redesigned and recoded.

Initially we performed empirical studies using custom-built models to make predictions for three different large industrial systems. We paid particular attention to selecting subject systems with very different functionalities, written in different languages, using different development paradigms to try to establish a sort of universal model that could make relatively accurate predictions for each of the systems.

Table 1. Percentage of Faults in Top 20% of Files for Six Systems

System	Years in Field	LOC in Last Release	Percentage Faults Identified
Inventory	4	538,000	83%
Provisioning	2	438,000	83%
Voice Response	2+	329,000	75%
Maintenance Support A	9	442,000	84%
Maintenance Support B	9	384,000	94%
Maintenance Support C	7	329,000	85%

The predictions were made using a negative binomial regression model, whose independent variables included static code characteristics and each file's change and fault history. Code characteristics were the size of the file, and the programming language. File history information included file age in terms of the number of previous releases the file had been in the system, and the number of changes and faults recorded in recent releases. We also investigated factors related to the number of developers who interacted with a file, including the cumulative number of distinct developers who had modified the file over its lifetime, and the number of developers who had modified it in the most recent release.

The model based on code characteristics and file history predicted the number of faults that would be associated with each file in the next release of the software. When the files were sorted in descending order of predicted faults, we found that the top 20% of the files typically contained well over 80% of the faults that were actually detected in those files.

The systems that were used as subjects of our first three empirical studies each contained hundreds of thousands of lines of code, and had histories ranging from two to four years in the field. Prediction results for these systems are summarized in the first three rows of Table 1. Detailed descriptions and prediction results for the inventory system and the provisioning system are presented in [2], and for the voice response system in [1].

2 Building a Tool

We have incorporated the defect prediction method into an automated tool to be used by developers and testers in the field. Our goals for the tool are that it should be very easy to learn and use, integrated into the normal development and testing environments, require no statistics background, rely only on data that is already collected for other purposes, run with reasonable speed (less than 1 minute for a large multi-release system), and present prediction results in an immediately understandable and useful format.

All the data used to make the predictions are extractable from the commercially-available change management/version control system used by each of the six projects that we have worked with to date. This change management system requires that a *modification request* (MR) be written in order for any changes to be made, whether the change is needed because a defect has to be fixed, because new functionality is being added in response to a change in the system specifications or requirements, or because regular maintenance updates are required.

Our initial prediction models were entirely hand-crafted, but during the fourth empirical study, whose subject was a business maintenance support system (Maintenance Support A), we compared the results of a custom-built model to a model whose coefficients were derived automatically using information learned from studying the three previous systems [3]. Somewhat surprisingly, the results from the automatable model were slightly better than those of the custom-built model for this system, and comparable to the custom-built results for the inventory system and the provisioning system. They also improve significantly on the custom results for the voice response system. The automatable model performed even better for two additional large components (B & C) of the business maintenance system. The automatable model's results for these three systems are shown in rows 4, 5, and 6 of Table 1.

These observations encouraged us to build a fully-automated tool using this general negative binomial regression model. For our initial research, we used shell scripts to do the data extraction, and to drive the prediction model. The models were fit using the Genmod procedure of SAS/STAT Release 8.01 [5]. We used this approach for the empirical study of Maintenance Support A, and began work on an integrated and robust tool design. For this first prototype, we used R [4] to fit the models and produce the predictions. For the current tool version, we wrote a custom C program to fit the negative binomial regression model to the data, and to apply the derived model to the extracted data of the release to be predicted. The custom C program significantly improved performance, and also avoids any issues that may arise from embedding a commercial product like SAS, or using a potentially unstable or evolving open-source product like R.

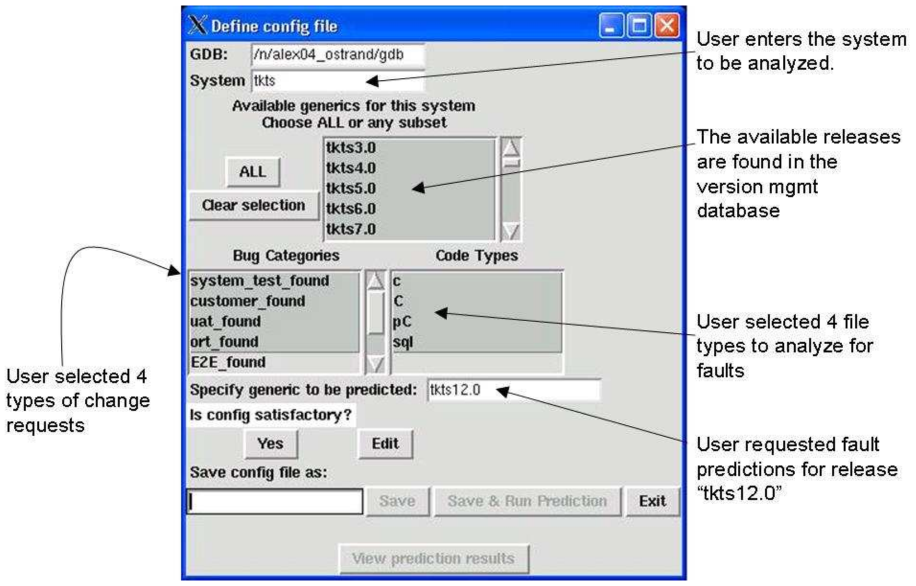
3 Using the Tool

Key considerations for the prediction tool are ease of use, and speed of execution. Because almost all expected users will be neither familiar with nor interested in the statistical foundation of the prediction model, the mathematics of the prediction is hidden under the hood.

The prototype tool has a simple GUI interface, with one screen used to define the parameters needed to create and run the model, and a second screen that presents prediction results to the user.

On the *Configuration Screen* shown in Figure 1, the user provides the following information to define the model and request a specific prediction.

– a pointer to the root node of the system's version management tree (GDB)
– the system to be analyzed
– a list of the system's previous releases, whose defect and change history will be used to create the defect prediction model.
– the types of change requests that should be considered defects



GDB: /n/alex04_ostrand/gdb
System tkts

tkts3.0
tkts4.0
tkts5.0
tkts6.0
tkts7.0

system_test_found
customer_found
uat_found
ort_found
E2E_found

c
C
pC
sql

Specify generic to be predicted: tkts12.0

User enters the system to be analyzed.

The available releases are found in the version mgmt database

User selected 4 types of change requests

User selected 4 file types to analyze for faults

User requested fault predictions for release "tkts12.0"

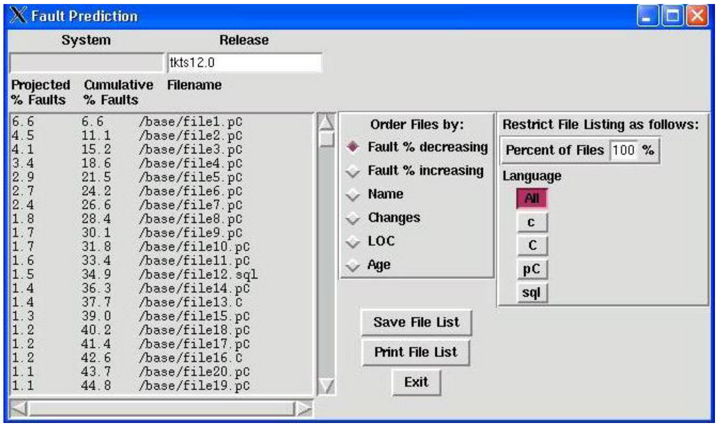
Fig. 1. Defining the configuration for predictions

– the file types for which the user expects fault-proneness predictions
– the single specific release for which the user desires fault-proneness predictions

The tool can provide default values for all of these except the name of the system to be analyzed and the specific release to be predicted. The list of file types allows the user to exclude non-executable types such as doc, jpg or wav, for which the predictions would not be meaningful, while including all relevant executable languages for the system such as C, C++, Java, SQL, as well as whatever special-purpose executable languages the project may use. When the configuration details are satisfactory, the user can save the configuration, and then run the prediction model.

Figure 2 shows the *Result Screen*, with a list of all files of the selected types in decreasing order of their predicted faults in the chosen release. The cumulative percent of faults allows the user to easily see how many and which files are needed to include any given percent of the predicted faults. In the example of Figure 2, the first fifteen files are predicted to contain 39.0% of the faults, while these fifteen files represent less than 1% of the total files in the system. In our case studies of the six systems in Table 1, we have frequently seen similarly top-heavy fault distributions.

The results screen presents the user with several options for displaying the predictions. The files can be ordered either by decreasing or increasing fault percentage, by file name, number of changes made in the prior release, file size(LOC), or file age. The user can also choose to display only files of any individual type (in the example: c, C, pC, sql), or only a certain percentage of the files.



System

Release tkts12.0

Percent of Files 100 %

```
Projected  Cumulative  Filename
% Faults   % Faults
6.6        6.6         /base/file1.pC
4.5        11.1        /base/file2.pC
4.1        15.2        /base/file3.pC
3.4        18.6        /base/file4.pC
2.9        21.5        /base/file5.pC
2.7        24.2        /base/file6.pC
2.4        26.6        /base/file7.pC
1.8        28.4        /base/file8.pC
1.7        30.1        /base/file9.pC
1.7        31.8        /base/file10.pC
1.6        33.4        /base/file11.pC
1.5        34.9        /base/file12.sql
1.4        36.3        /base/file14.pC
1.4        37.7        /base/file13.C
1.3        39.0        /base/file15.pC
1.2        40.2        /base/file18.pC
1.2        41.4        /base/file17.pC
1.2        42.6        /base/file16.C
1.1        43.7        /base/file20.pC
1.1        44.8        /base/file19.pC
```

Fig. 2. Prediction of defect frequency for files

4 Conclusions

Although a number of other research groups have been developing prediction models, none of the others has validated them by performing the number of large industrial empirical studies we have, nor have they followed systems for many releases. In total, we have made predictions for well over 100 distinct releases in six different systems. In addition, we have built an automated tool to make the predictions, enabling testers and developers to use the technology in a practical software development environment. Again, to our knowledge, no other group has accomplished this. Currently, the tool has been integrated into a widely-used version control/change management system and is about to become available to production projects to use to help them make their testing more efficient and effective.

References

1. Bell, R.M., Ostrand, T.J., Weyuker, E.J.: Looking for Bugs in All the Right Places. In: Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA 2006), Portland, Maine, July 2006, pp. 61–71 (2006)
2. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans. on Software Engineering 31(4) (April 2005)
3. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Automating Algorithms for the Identification of Fault-Prone Files. In: Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA 2007), London, England (July 2007)
4. The R Project for Statistical Computing, `http://www.r-project.org/`
5. SAS Institute Inc. SAS/STAT 9.1 User's Guide, SAS Institute, Cary, NC (2004)

SeeCode – A Code Review Plug-in for Eclipse

Moran Shochat, Orna Raz, and Eitan Farchi

IBM R&D Labs in Israel, Haifa, 31905, Israel
{morans,ornar,farchi}@il.ibm.com

Abstract. It is well known that code reviews are among the most effective techniques for finding bugs [2, 3, 4]. In this paper, we describe a code review tool, SeeCode, which supports the code review process. SeeCode is an Eclipse plug-in and thus naturally integrates into the developer's working environment. It supports a distributed review environment and the various roles used in a review meeting. Reviewers can review the code at the same time, either through a virtual or a face-to-face meeting, or at different times. Review comments and Author navigation through the code are visible to all reviewers. Review comments are associated with line numbers, and the association is maintained when the code is changed by the developer. The integration with the Eclipse [8] Integrated Development Environment (IDE) enables easy code navigation, which is required especially when object-oriented code is reviewed.

SeeCode also supports a quantitative feedback mechanism that reports the effectiveness of the ongoing review effort. This feedback is updated as the review progresses, and can be utilized by the review moderator to keep the review process on track.

SeeCode has been piloted by several IBM groups with good feedback. The distributed review feature and integration with the IDE are particularly noted by users as key features.

1 Introduction

In this paper we describe SeeCode, a plug-in for Eclipse that supports the code review process. Code review is the most effective technique for ensuring software quality [2, 3, 4]. In particular, the selective homeworkless review technique that we described in a previous paper [1] enables effective code review in time-constrained and possibly distributed projects. In selective homeworkless review, the artifacts for review are selected according to quality concerns and review methodologies with little or no preparation are used.

We have been conducting code reviews with a large number of diverse development teams across IBM for several years. Our experience indicates that an effective review tool should fulfill the following requirements:

a) Enable the review of source files and provide a mechanism for handling and storing review comments
b) Support the review meeting process and roles, possibly for a distributed team
c) Provide syntax highlighting and easy code navigation (e.g., jumping from function usage to its definition)

d) Maintain the association between comments and source lines, even when the code is changed
e) Provide ongoing feedback on the effectiveness of the review process
f) Allow revision control of the review comments along with the source files

SeeCode was designed to meet the above list of requirements as well as to allow various review methodologies.

Other code review tools exist. However, each of these tools fulfills only part of the above requirements. Jupiter [5], like SeeCode, is implemented as an Eclipse plug-in and thus naturally integrates into the developer working environment, taking advantage of Eclipse features such as syntax highlighting, code navigation and association of markers to source lines. However, Jupiter only supports a single user and therefore does not fulfill the requirement for review roles and team distribution support. Codestriker [6] is a web application that supports a distributed review environment in which all reviewers see the same source file and its associated review comments. However, Codestriker does not provide substantial navigation support, nor does it provide an association of review comments with the underlying code when the code changes. Moreover, SeeCode provides improved distributed review support. SeeCode supports the review roles of Owner, Scribe and Reviewers and each of these roles may be executed on separate computers. This supports both virtual and face-to-face review meeting modes. SourcePublisher [7] generates printouts of the code with syntax highlighting and links to other relevant parts of the code. However, SourcePublisher provides no comment handling mechanism, nor does it provide support for the review process.

A novel feature of SeeCode is the integrated quantitative feedback mechanism that reports the effectiveness of the ongoing review effort. This feedback is updated as the review progresses and can be utilized by the review moderator to keep the review process on track.

SeeCode has been piloted by several IBM groups with good feedback. The distributed review feature and the integration with the IDE are particularly noted by users as being key features.

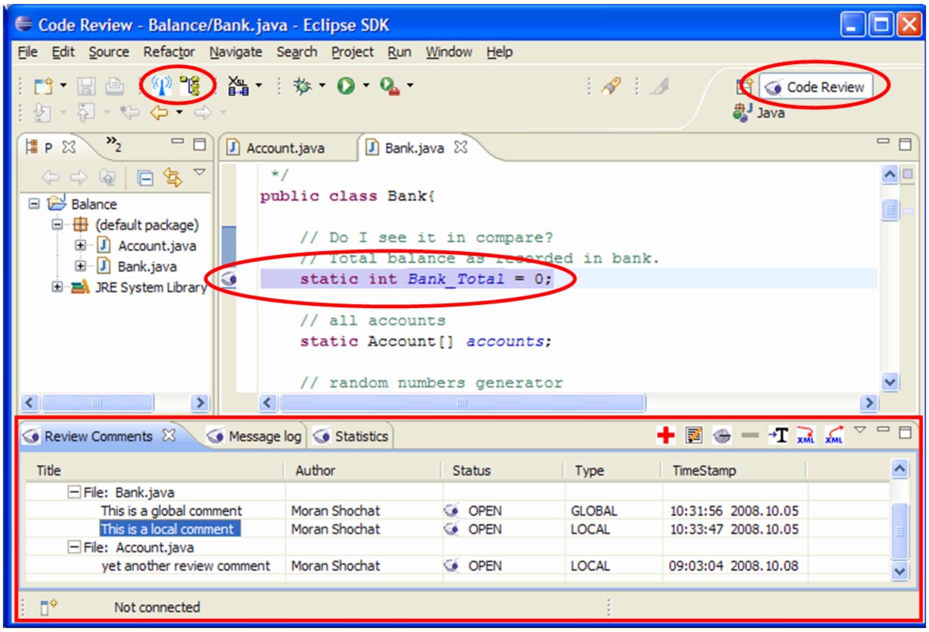
SeeCode is a beta research tool. We are still implementing additional features and improving the tool according to user feedback.

2 SeeCode Main Features

2.1 Code Review Perspective

The SeeCode plug-in adds the ‘Code Review’ perspective to the Eclipse workbench (see Fig. 1, below). The Code Review perspective includes three views – the ‘Review Comments’ view displays all review comments, the ‘Message Log’ view displays a log of messages received when the tool is in distributed mode, and the ‘Statistics’ view displays statistics on the review process.

As an Eclipse plug-in, SeeCode naturally integrates into the IDE and enhances the review process with built-in Eclipse features such as syntax highlighting and easy code navigation. Code navigation is required especially when object-oriented code is reviewed, since frequent jumps from one method to another are required. SeeCode uses Eclipse markers to store the review comments, and thus allows the persistent association of comments to source lines even when the source is changed by the author.



```
 */
public class Bank{

    // Do I see it in compare?
    // Total balance as recorded in bank.
    static int Bank_Total = 0;

    // all accounts
    static Account[] accounts;

    // random numbers generator
```

Title	Author	Status	Type	TimeStamp
File: Bank.java				
This is a global comment	Moran Shochat	OPEN	GLOBAL	10:31:56 2008.10.05
This is a local comment	Moran Shochat	OPEN	LOCAL	10:33:47 2008.10.05
File: Account.java				
yet another review comment	Moran Shochat	OPEN	LOCAL	09:03:04 2008.10.08

Fig. 1. ‘Code Review’ perspective. ‘Review Comments’ view, ‘Message Log’ view and ‘Statistics’ view are placed at the bottom of the Eclipse workbench. Comments are marked by a SeeCode icon in the editor ruler bar.

2.2 Distributed Review

SeeCode defines three review roles - the code Owner, the meeting Scribe and the Reviewer(s). Each SeeCode user has one of the above roles and is running on an independent computer. The Scribe and the Reviewers connect to the Owner, allowing them to view the source files, the review comments and the navigation actions of the Owner explaining the code. All reviewers can still independently navigate in their own IDEs. This supports, for example, the common usage of looking at a related piece of code individually to verify a possible issue before stating it.

SeeCode enables the review of code either in a *team meeting* or in an *asynchronous* review mode. In a review meeting (either face-to-face or virtual) the Owner explains the code; the Reviewers suggest comments and the Scribe documents them using SeeCode. In an asynchronous review, the Owner selects the files for review and the Reviewers connect at different times, receive the up-to-date list of comments, and add their own comments. Often, the asynchronous review is used as a preparation stage for the review meeting.

Even though SeeCode supports the above-mentioned roles and modes, it was designed not to enforce them. This makes SeeCode more flexible and enables its use with various code review techniques.

2.3 Review Comments

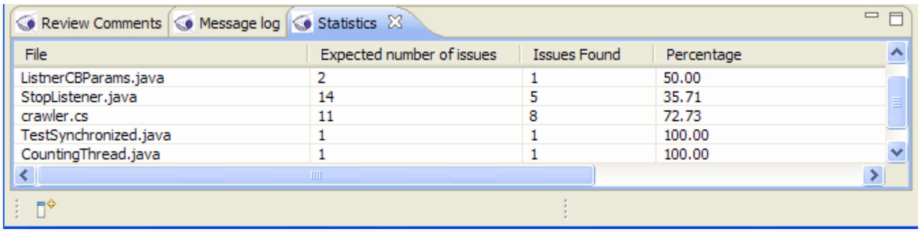
SeeCode allows the addition of *local* and *global* comments. Local comments are associated with a certain line in the source file while global comments are associated with the entire file and are used for general issues. Existing comments are marked by a SeeCode icon in the editor ruler bar (see Fig. 1, above) and can be further edited, replied to or closed.

SeeCode review comments are saved in an XML file. There is a comments file for each Eclipse project in the user workspace. Using any version control system, the comments file can be controlled together with the project source files to maintain the correct association between comments and line numbers.

In addition, SeeCode enables selective export of comments into text or XML files and selective import of comments from previously exported XML files. This is useful when reviewing the code without connecting to the distributed review. In such cases, reviewers export their comments and send them to the Owner, who merges them with his comments file.

2.4 Statistics View

The 'Statistics' view provides a novel quantitative feedback mechanism that reports the effectiveness of the ongoing review effort. It displays the updated number of comments for each file and compares it to the expected number of issues (see Fig. 2, below). The expected number of issues is based on our experience with several IBM teams suggesting an average review rate of about 100 lines per hour and an average issue recording rate of 2-3 per person hour [1]. This feedback can be utilized by the review moderator to keep the review process on track.



File	Expected number of issues	Issues Found	Percentage
ListnerCBParams.java	2	1	50.00
StopListener.java	14	5	35.71
crawler.cs	11	8	72.73
TestSynchronized.java	1	1	100.00
CountingThread.java	1	1	100.00

Fig. 2. 'Statistics' view

3 Deployment of SeeCode

SeeCode has been piloted by several IBM groups with good feedback, including some of the following responses:

a) Using the tool is intuitive and easy to learn
b) Distributed review is very useful for teams that are spread over different locations and time zones
c) Syntax highlighting and code navigation features of Eclipse help the reviewers in understanding the code, making the review more efficient

d) Performing the review inside the IDE saves time and extra effort (e.g., no need to send files for review and no need to document the comments in a separate tool)

Even though SeeCode is deployed together with our effective review methodology (Selective Homeworkless Reviews [1]), it does not restrict usage to a specific methodology. We saw a good example of this with a team that is distributed over several IBM locations. This team decided to use SeeCode's asynchronous review mode for several days prior to the review meeting as a preparation stage.

A noteworthy limitation of SeeCode is the fact that it is less useful for teams developing in a programming language that has no Eclipse plug-in. This was the case for an IBM team that develops PL.8 code. In such cases, we recommend performing the reviews in the environment that the team is used to, without SeeCode.

4 SeeCode Planned Features

As SeeCode development moves forward, new features are planned. Planned features include implementing the integration of other data into SeeCode and providing supporting views to assist in:

a) Focusing the review according to external data such as file change history, static analysis report or code coverage report
b) Focusing the review according to a specific programming concern such as performance, reliability, concurrency, etc
c) Focusing the review on code changes by performing the review in compare view

References

1. Farchi, E., Ur, S.: Selective Homeworkless Review. In: International Conference on Software Testing, Verification and Validation, pp. 404–413 (2008)
2. Porter, A.A., Siy, H.P., Votta, L.G.: A review of software inspections. Advances in Computers 42, 39–76 (1996)
3. Wiegers, K.E.: Read my lips: No new models! IEEE Software 15(5), 10–13 (1998)
4. Fagan, M.: A history of software inspections. In: Software pioneers: contributions to software engineering, pp. 562–573. Springer, New York (2002)
5. Jupiter – An Eclipse plugin for code review (accessed, July 2008), `http://csdl.ics.hawaii.edu/Plone/research/jupiter`
6. Codestriker: collaborative code reviewer (accessed, July 2008), `http://codestriker.sourceforge.net/`
7. Source Publisher (accessed, July 2008), `http://www.scitools.com/products/sourcepublisher/`
8. Eclipse (accessed, July 2008), `http://www.eclipse.org/`

User-Friendly Model Checking: Automatically Configuring Algorithms with RuleBase/PE

Ziv Nevo

IBM Haifa Research Lab, Haifa, Israel
nevo@il.ibm.com

Abstract. Model checking is known to be computationally hard, meaning no single algorithm can efficiently solve all problems. A possible approach is to run many algorithms in parallel until one of them finds a solution. This approach is sometimes called state-of-the-art (SOTA) model checker. However, hardware resources are often limited, forcing some selection. In this paper we present an automatic decision system, called Whisperer, which generates an optimized set of configured algorithms for a given model-checking problem. The system weights the advice of advisors, each predicting the fitness of a different algorithm for the problem. Advisors also monitor the progress of currently running algorithms, allowing the replacement of ineffective algorithms. Whisperer is built into the formal verification platform, RuleBase/PE, and allows novice users to skip the delicate task of algorithm selection. Our experiments show Whisperer, after some training, performs nearly as well as SOTA.

1 Introduction

A significant number of model-checking algorithms have been suggested over the years. Finding an efficient algorithm for solving a given model-checking problem may sometimes be non-trivial. In fact, for hard problems it may be non-trivial to find **any** configured algorithm that solves them. Running all algorithms in parallel until one of them finds a solution may prove useful on such cases. This approach is sometimes referred to as state-of-the-art (SOTA) model checker [1].

With the multitude of algorithms, the SOTA approach may require too many CPUs. Expert engineers therefore tend to run a few problems at the beginning of each project to get a feel for the best algorithms and their best settings for the current hardware design. They then use this set of algorithms along the project. Still, some problems may defy this set, and require further experimentation.

This paper proposes a decision system, called Whisperer, which automatically suggests a set of configured model checkers for a given problem. This automation allows hiding complex algorithmic details and making more friendly tools.

Expert systems are already fundamental to theorem provers, e.g., HOL [2] and PVS [3]. Changing strategies inside a model checker was also proposed [4], as well as using machine learning techniques for solver tuning [5]. An expert system was suggested to guide the flow of a transformation-based verification system [6], and to select a best-fit SAT or QBF solver [7,8]. Whisperer is different from these in its attempt to provide a **set** of complementing model checkers to run in parallel and in its online algorithm of weighting expert advice.

RuleBase/PE is IBM's industrial strength formal-verification platform, replacing its predecessor, RuleBase [9]. In addition to offering a larger set of more advanced model-checking algorithms (a.k.a. *engines*), RuleBase/PE allows the parallel execution of several engines on a single problem, thus realizing the SOTA approach. Whisperer is integrated into RuleBase/PE.

2 Whisperer – Predicting Best-Fit Model Checkers

Whisperer is a decision system for dynamically choosing an effective set of model-checking engines for a given problem. The set of engines it may select from includes a guided explicit model checker, BDD-based engines (forwards and backwards reachability, partial search, abstraction refinement) and SAT-based engines (BMC, abs. ref., overapproximated reachability). Whisperer's decisions are based on problem parameters (flip-flop count, property type, etc.), on past performance (same problem and problems in the same design) and on user settings (process limit, expected result). Whisperer records engines performance for future analysis, assuming most problems in a given project behave similarly[1].

2.1 Advisors

Whisperer is implemented as an algorithm for combining expert advice. Each engine is assigned an *advisor*. Advisors predict engine fitness for the given problem and configure engine settings. A configured engine together with a fitness score (a number between 0 and 5) is called an *advice*. In addition, advisors monitor the progress of their advice (when running), and may adjust fitness accordingly.

Advisors are implemented as C++ inherited classes, allowing modularity and maintainability. Each advisor implements methods for getting its advice, for evaluating a currently running advice and for drawing conclusions from terminated advice. The implementation is usually written by engines' developers, using an API for querying relevant data. A simplistic example advisor is shown in Fig. 1.

In the `getAdvice` method advisors generate their advice. The example code shows how settings may be set and fitness may be adjusted in response to user policy and problem parameters. Whisperer advisors mainly consider these two inputs. For example, BDD-based engines tend to be sensitive to FF count, explicit engines prefer a small degree of non determinism, BMC is useless when attempting proofs. The advisors also use weighted random decisions for choosing a value for a specific setting. Weights for each value change according to past performance, allowing learning the best-performing configuration. Advisors for related algorithms (e.g., SAT-based) share the best values for common settings.

In the `evaluate` method advisors monitor currently running advice, and change fitness as they see fit. The example advisor uses this method to gradually

[1] This is an empirical observation, made by our users. It is probably due to chunks of logic, common to most problems, that make some engines outperform others.

```
Advice *ExAdvisor::getAdvice() {
    Advice *ad = new Advice(this);
    if (getUserPolicy() == FALSIFY) ad->SetSetting("Falsify", "true");
    ad->currFitness = m_baseFitness;
    if (getNumStateVars() > 500) ad->currFitness -= 1.;
    return ad;
}
void ExAdvisor::evaluate(Advice *ad) {
    ad->currFitness = m_baseFitness - ad->timeSinceLastReport()/60.;
}
void ExAdvisor::drawConclusions(Advice *ad) {
    if (ad->solvedProblem()) m_baseFitness += 0.1;
}
```

Fig. 1. An example advisor

decrease advice fitness as time passes since the last reported progress. Whisperer advisors also analyze the content of progress reports and take into account the approximated problem hardness. For example, the advisor for the SAT-based BMC maintains high fitness as long as "bounded passed" results keep coming.

In their `drawConclusions` method advisors should learn from the performance of their advice. The example advisor awards itself whenever it solves a problem by increasing its base fitness. Whisperer advisors usually change weights of specific values in accordance with the overall progress achieved by their engine.

Whisperer also allows advisors to store arbitrary data on disk, to be used in future verification sessions. Data is usually loaded in the advisor's constructor and saved in its destructor. Thus, the example advisor may save its base fitness value and provide higher fitness scores following a series of successes. Whisperer advisors also save various weights and problem statistics produced by the engine.

2.2 Advice Selection

Having a set of advisors as described above, Whisperer's job then is to decide which advice to take, which to reject and which to replace. Whisperer should not only count on the learning done by advisors, but should rather also learn for itself which advisors correctly predict the fitness of their advice. This allows Whisperer, to some extent, recovering from unpredictable cases, not anticipated by the engine developers. Whisperer works iteratively once every few seconds as follows.

```
while (problem is unsolved)
    newAdvices = getNewAdvices();
    reEvaluateRunningAdvices(runningAdvices);
    killSomeRunningAdvices( runningAdvices, newAdvices );
    addPromisingNewAdvices( runningAdvices, newAdvices );
```

After getting an advice from each advisor, each advice is assigned a mark, being the product of the advice's fitness and the advisor's *reliability factor*. The reliability factor is a number, assigned to each advisor, which reflects how successful the advisor is in predicting the fitness of its advice in the current project.

Whisperer then asks its advisors to re-evaluate the fitness of their currently running engines, as engines progress reports may change it. Poor evaluations together with attractive new advice may terminate a running engine and replace it with another. The decision is a weighted random decision, based on the engine runtime and on the marks difference compared to the best new advice. An advice running for a long time has better chances to be replaced.

Finally, advice is repeatedly picked in a classical weighted random selection according to its mark, until the number of engines meets the process limit.

Whenever an engine is terminated, the corresponding advice is assessed and its advisor is rewarded accordingly. Good advice, correctly predicting engine performance, increases its advisor's reliability by a factor depending on the engine runtime and whether or not the engine solved the problem. Bad advice, predicting good fitness for a poorly performing engine decreases reliability by a factor depending on the engine runtime and on the wrongly predicted fitness.

Reliability factors are loaded from disk at the beginning of each model-checking session (when starting a new project, all advisors get the same factor), and are saved to disk when it ends. Thus, the more problems are attempted, the more Whisperer becomes tuned to the current formal verification project.

3 Experimental Results

We compare running RuleBase/PE with Whisperer to running many engines in parallel as in SOTA model checker. We used a set of industrial examples, each being one real-life model-checking problem of a distinct hardware design.

For each example we made one run with 16 parallel engines (some engines had multiple, differently configured instances). This is not as exhaustive as SOTA, because not **every** possible engine configuration was covered. Covering them all would have required hundreds of processes. Yet, it is a reasonable approximation, as we chose the most productive configurations to the best of our knowledge. We also ran Whisperer twice for each example. First without any training, then after allowing Whisperer to solve five random model-checking problems of the same hardware design. All Whisperer runs used a maximum of 4 concurrent processes.

We used dual-processor, dual-core 2.4GHz AMD Opteron servers with 8GB of RAM, running under a load-balancing system. Results are shown in Table 1. The table shows for each example its size (number of flip-flops after model reductions) and runtimes for our SOTA approximation and for Whisperer before and after training. An asterisk indicates Whisperer chose SOTA's best performing engine, but not necessarily the best configuration for this engine.

Results demonstrate that Whisperer is able to compete with our approximation of SOTA, and to usually improve after a short training. On most cases the trained Whisperer chose the best performing engine. On one example (B) the trained Whisperer chose a better performing configuration than what we thought to be optimal. For the rest of the cases Whisperer usually came with a reasonable alternative engine, performing nearly as good as SOTA, but using significantly less concurrent processes.

Table 1. Whisperer vs. SOTA (runtimes are in seconds)

Example	*Size*	SOTA (16 CPUs)	Whisperer (4 CPUs)	Trained Whisperer (4 CPUs)
A	755	157	173	168
B	4996	208	333*	192*
C	5158	383	462	384*
D	386	70	158	203
E	550	314	337	344
F	2025	116	184*	142*
G	2391	74	1027	84*
H	1418	190	190*	199*
I	15113	874	973	882*
J	140	265	714	265*

4 Conclusion

This paper describes Whisperer, an automatic system for selecting and configuring a set of model-checking algorithms to run in parallel. Whisperer uses the concept of weighting expert advice, having each algorithm being represented by an expert. Adjusting expert weight according to its performance, allows Whisperer to successfully assign a useful set of algorithm for each problem.

Future research may allow Whisperer to learn from human experts, and to analyze relationships between the various algorithms. For example, algorithms with correlated success should not run together.

References

1. Narizzano, M., Pulina, L., Tacchella, A.: The 3rd QBF solvers comparative evaluation. Journal on Satisfiability, Boolean Modeling and Computation, 145–164 (2006)
2. Gordon, M.: Mechanizing programming logics in higher order logic. In: Current Trends in Hardware Verification and Automated Theorem Proving, pp. 387–439. Springer, Heidelberg (1989)
3. Srivas, M., Rueß, H., Cyrluk, D.: Hardware verification using PVS. In: Kropf, T. (ed.) Formal Hardware Verification. LNCS, vol. 1287, pp. 156–205. Springer, Heidelberg (1997)
4. Shacham, O., Yorav, K.: Adaptive application of SAT solving techniques. In: Proc. 3rd International Workshop on Bounded Model Checking, pp. 35–50 (2005)
5. Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting verification by automatic tuning of decision procedures. In: FMCAD, pp. 27–34 (2007)
6. Mony, H., Baumgartner, J., Paruthi, V., Kanzelman, R., Kuehlmann, A.: Scalable automated verification via expert-system guided transformations. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 159–173. Springer, Heidelberg (2004)
7. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
8. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 574–589. Springer, Heidelberg (2007)
9. Beer, I., Ben-David, S., Eisner, C., Landver, A.: RuleBase: an industry-oriented formal verification tool. In: Proc. 33rd DAC, pp. 655–660 (1996)

Author Index