Gordon Fraser
Angelo Gargantini (Eds.)

LNCS 6143

# Tests and Proofs

4th International Conference, TAP 2010
Málaga, Spain, July 2010
Proceedings

Springer

Gordon Fraser   Angelo Gargantini (Eds.)

# Tests and Proofs

4th International Conference, TAP 2010
Málaga, Spain, July 1-2, 2010
Proceedings

Springer

Volume Editors

Gordon Fraser
Saarland University, Computer Science
Saarbruecken, Germany
E-mail: fraser@cs.uni-saarland.de

Angelo Gargantini
University of Bergamo, DIIMM
Dalmine, Italy
E-mail: angelo.gargantini@unibg.it

Library of Congress Control Number: 2010929258

CR Subject Classification (1998): D.1, F.3, D.2, D.3, D.2.4, F.4.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-13976-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13976-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

# Preface

This volume contains the proceedings of TAP 2010, the 4th International Conference on Tests and Proofs held during July 1–2 in Málaga, Spain as part of TOOLS Federated Conferences.

TAP 2010 was the fourth event of an ongoing series of conferences devoted to the convergence of proofs and tests. In the past, proving and testing were seen as very different and even competing techniques. Proving people would say: If correctness is proved, what do we need tests for? Testers, on the other hand, would claim that proving is too limited in applicability and testing is the only true path to correctness. Of course, both have a point, but to quote Ed Brinksma from his 2009 keynote at the Dutch Testing Day and Testcom/FATES:

> "Who would want to fly in an airplane with software proved correct, but not tested?"

Indeed, the true power lies in the combination of both approaches. Today, modern test systems rely on techniques deeply rooted in formal proof techniques, and testing techniques make it possible to apply proof techniques where there was no possibility previously.

At a time when even mainstream software engineering conferences start featuring papers with both "testing" and "proving" in their titles, we are clearly on the verge of a new age where testing and proving are not competing but finally accepted as complementary techniques. Albeit, we are not quite there yet, and so the TAP conferences aim to provide a forum for researchers working on the converging topics and to raise general awareness of this convergence.

In 2007 the first TAP conference was held at ETH Zürich with Bertrand Meyer as Conference Chair and Yuri Gurevich as Program Chair. The idea of this new conference was to provide a forum for the cross-fertilization of ideas and approaches from the testing and proving communities. Following its success, TAP 2008 was held at the Monash University Prato Centre near Florence, Italy, and TAP 2009 was held in Zürich again. As in 2009, the 2010 edition was again part of TOOLS Federated Conferences, a set of related conferences offering a combined venue.

Our sincere thanks go to the authors who submitted their work for consideration at this conference. We would like to thank the Conference Chairs Bertrand Meyer and Yuri Gurevich for their work on this and previous TAP conferences. A big thank you also to the Program Committee members and external reviewers, whose work is essential in establishing a high-quality conference. Thanks also go to the TOOLS Federated Conferences organizers and particularly Antonio Vallecillo, who took care of all the local arrangements and negotiations with Springer. Finally, thanks also to Easychair, which makes the life of program chairs so much easier.

For TAP 2010, we selected 10 full papers and 2 short papers, which are included in this volume. In total 20 submissions were made, and each paper was reviewed by at least three reviewers, followed by a lively discussion phase, assuring that only papers of high quality were accepted. We are also very proud that the conference featured keynotes by Mike Ernst (University of Washington) and Nachi Naggappan (Microsoft Research); the abstracts of these talks are included in this volume. In sum, this made up a terrific program, and we hope that everyone will enjoy reading the proceedings.

April 2010

Gordon Fraser
Angelo Gargantini

# Conference Organization

## Conference Chairs

| | |
|---|---|
| Bertrand Meyer | ETH Zürich, Switzerland |
| Yuri Gurevich | Microsoft Research, USA |

## Program Chairs

| | |
|---|---|
| Gordon Fraser | Saarland University, Germany |
| Angelo Gargantini | University of Bergamo, Italy |

## Program Committee

| | |
|---|---|
| Bernhard K. Aichernig | Graz University of Technology, Austria |
| Paul Ammann | George Mason University, USA |
| Bernhard Beckert | University of Koblenz, Germany |
| Dirk Beyer | Simon Fraser University, Canada |
| Koen Claessen | Chalmers University of Technology, Sweden |
| John A. Clark | University of York, UK |
| Catherine Dubois | ENSIIE, Evry, France |
| Carlo Furia | ETH Zürich, Switzerland |
| Patrice Godefroid | Microsoft Research, USA |
| Martin Gogolla | University of Bremen, Germany |
| Arnaud Gotlieb | IRISA, France |
| Reiner Haehnle | Chalmers University of Technology, Sweden |
| Bart Jacobs | Katholieke Universiteit Leuven, Belgium |
| Gregory M. Kapfhammer | Allegheny College, USA |
| Victor Kuliamin | Russain Academy of Sciences, Russia |
| Karl Meinke | KTH Royal Institute of Technology, Sweden |
| Manuel Nunez | Universidad Complutense de Madrid, Spain |
| Sam Owre | SRI International, USA |
| Doron Peled | Bar Ilan University, Israel |
| Wolfram Schulte | Microsoft Research, USA |
| Yannis Smaragdakis | University of Massachusetts, USA |
| Assia Touil | Supélec, France |
| T.H. Tse | University of Hong Kong, China |

## External Reviewers

| | |
|---|---|
| Cesar Andres | Universidad Complutense de Madrid, Spain |
| Emil Axelsson | Chalmers University of Technology, Sweden |
| Matthieu Carlier | IRISA, France |
| Christoph Gladisch | University of Koblenz, Germany |
| Lars Hamann | University of Bremen, Germany |
| Nadjib Lazaar | IRISA, France |
| Ann Lillieström | Chalmers University of Technology, Sweden |
| Mercedes Merayo | Universidad Complutense de Madrid, Spain |
| Ina Schaefer | Chalmers University of Technology, Sweden |
| Nicholas Smallbone | Chalmers University of Technology, Sweden |

## Local Organization

### Organizing Chair

| | |
|---|---|
| Antonio Vallecillo | University of Málaga, Spain |

### Workshops Chair

| | |
|---|---|
| Francisco Duran | University of Málaga, Spain |

### Registration Chair

| | |
|---|---|
| Carlos Canal | University of Málaga, Spain |

### Financial Chair

| | |
|---|---|
| Ernesto Pimentel | University of Málaga, Spain |

### Venue/Local Issues

| | |
|---|---|
| Manuel F. Bertoa | University of Málaga, Spain |
| José M. Álvarez | University of Málaga, Spain |

### Organizing Team

| | |
|---|---|
| Daniel Ruiz | University of Málaga, Spain |
| Eduardo Rivera | University of Málaga, Spain |
| Fernando López | University of Málaga, Spain |
| Javier Troya | University of Málaga, Spain |
| José Antonio Martín | University of Málaga, Spain |
| Jose Bautista | University of Málaga, Spain |
| Lola López | University of Málaga, Spain |

# Table of Contents

# How Tests and Proofs Impede One Another: The Need for Always-On Static and Dynamic Feedback

Michael D. Ernst

Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
mernst@cs.washington.edu

**Abstract.** When creating software, developers rely on feedback in the form of both tests and proofs. The tests provide dynamic feedback from executing the software. The proofs provide static, conservative feedback from analyzing the source code. Testing is widely adopted in practice, but type systems are the only variety of proof to achieve widespread adoption.

Dynamic and static feedback provide complementary benefits, and neither one dominates the other. Sometimes, sound global static checking is most useful. At other times, running tests is most useful. Unfortunately, current languages and IDEs impose too rigid a model of the development process. As a result, the developer is not in control of the development process. In situations where a small amount of appropriate feedback could have yielded insight, the developer must choose between either static or dynamic feedback, and warp his or her development style to the limitations of the language. This situation is wrong: the developer should always have access to immediate execution feedback, and should always have access to sound static feedback.

For example, in a typical statically-typed language, a developer is prevented from running the program if any type errors exist, even if they are not germane to the particular test. In a dynamically-typed (or weakly-typed) language, a developer is free to build insight by experimenting, but never gets the assurance of a proof of type correctness.

We need a better approach. A programmer should be able to view and execute a program through the lens of sound static typing. If the compiler issues no type errors, that is a guarantee (a proof) of static type soundness. A programmer should also be able to view and execute the same program through the lens of dynamic typing with no statically-imposed restrictions. The run-time system should suppress static type errors, unless they lead to user-visible failures, or the developer wishes to examine them. Furthermore, the programmer should be able to switch between these two views as often as desired, or to use them both simultaneously.

A typical use case is: a developer starts from an existing statically-typed codebase (or start writing new code), uses both dynamic and static feedback, and finally restores static type-correctness. Because the developer has types in mind and even uses the type system, large variations from statically-typeable idioms are unlikely.

The ability to execute and run code at any moment is useful in many circumstances, including software creation (e.g., prototyping) and software evolution (e.g., representation or interface changes, library replacement, exploratory changes). Approaches such as prototyping in a dynamic language then rewriting with types, or viewing one part of the program as typed and another part as untyped, address only a subset of these scenarios and impose a significant burden on the developer.

I will describe additional background, theory, and practical details. I will also share experience with an implementation, Ductile, that supports the goal of always-on static and dynamic feedback.

# Myths in Software Engineering: From the Other Side

Nachiappan Nagappan

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
nachin@microsoft.com

**Abstract.** An important component of Empirical Software Engineering (ESE) research involves the measurement, observation, analysis and understanding of software engineering in practice. Results analyzed without understanding the contexts in which they were obtained can lead to wrong and potentially harmful interpretation. There exist several myths in software engineering, most of which have been accepted for years as being conventional wisdom without having been questioned. In this talk we will deal briefly with a few popular myths in software engineering ranging from testing and static analysis to distributed development and highlight the importance of context and generalization.

**Keywords:** Empirical Software Engineering, Code coverage, Failures, People, development teams, Software Inspection, Distributed development, Software assertions.

## 1 Introduction

The ISERN[1] community (a group made up of different research labs, universities and industrial partners) states its purpose as, *"software engineering is a relatively new and immature discipline. In order to mature, we need to adopt an experimental view of research allowing us to observe and experiment with technologies, understand their weaknesses and strengths, tailor technologies to the goals and characteristics of particular projects, and package them together with empirically gained experience to enhance their reuse potential in future projects"*.

In empirical software engineering it is important to contextualize the environment in which the results are obtained to generalize results across studies. In general practitioners become more confident in a theory when similar findings emerge in different contexts [1]. Towards this end, empirical software engineering presents special emphasis on replication of experiments to learn and generalize results across different contexts, environments and domains.

Empirical Software engineering also focuses on evaluating oft held opinions in software engineering as some of these opinions are purely opinions and are not backed up by scientific data. In this keynote, I would like to shed some light on interesting case studies from industry that show some myths to be false, explain the importance of context and show the utility of building empirical bodies of knowledge.

[1] http://isern.iese.de/network/ISERN/pub/isern.manifesto.html

## 2 Case Studies

Some of the myths I plan to address in this talk with appropriate citations for further reading are:

**Code coverage:** Does higher code coverage mean better quality. Can we stop testing when we reach 100% code coverage? Are there better measures to quantify test efficacy? Should the goal be to maximize coverage? [2]

**Static analysis:** Static analysis tools are easy to use and low cost to deploy in software development teams. The downside being static analysis bug finding tools produce a lot of false positives. Are there better techniques for fault fix prioritization? When should we stop fixing static analysis bugs? What is the cost-benefit of utilizing static analysis defects? [3]

**Unit testing:** Is it really beneficial to do unit testing? Is there empirical evidence to show that teams adopting unit testing product better quality products? [6]

**Test Driven Development:** Test Driven Development (TDD) is a technique proposed as part of the Agile software development practices like Extreme Programming (XP). Does TDD produce better quality code? What is the cost-benefit trade-off of doing TDD? [7]

**Inspections:** Inspections have been widely practiced in industry. An important fact associated with inspections is that they are only as good as the inspector involved in the process. How does the human element influence the result of inspections? [8]

**Assertions:** Software assertions have often been discussed as an important best practice in software development. Has there been any empirical evidence of using assertions in software development that improved the efficacy of the project? [9]

**Distributed development:** We are increasingly observing large commercial projects being built by teams distributed across the world. We see professionals on either side of the spectrum who strongly argue for or against distributed software development. Does distributed software development affect quality? Do distributed teams work on simpler tasks and involve more communication overhead? [4]

**Minor contributors:** Software development is a task that involves a variety of individuals working together. Do certain individuals have a higher likelihood of causing failures/problems in the system than others?

**Socio-technical networks:** On a related note, software development is an activity that involves teams of developers coordinating with each other. In this regard there are multiple dependency structures which exist (due to the code dependencies/people dependencies etc). Does the intersection of these dependencies predict failures? [5]

**Organizational metrics:** Conway's law (organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations [10] ) has been used often in software organizations to organize teams. How important is Conway's law and what are its implications on software quality? [11]

## References

1. Basili, V.R., Shull, F., Lanubile, F.: Building Knowledge Through Families of Experiments. IEEE Transactions on Software Engineering 25(4), 456–473 (1999)
2. Mockus, A., Nagappan, N., Dinh-Trong, T.T.: Test Coverage and Post-verification Defects: A Multiple Case Study. In: Empirical Software Engineering and Measurement (ESEM), Orlando, FL (2010)
3. Nagappan, N., Ball, T.: Static Analysis Tools as Early indicators of Pre-Release Defect Density. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, Springer, Heidelberg (2006)
4. Bird, C., et al.: Does Distributed Development affect Software Quality? An empirical Case Study of Windows Vista. In: International Conference on Software Engineering (ICSE), Vancouver, Canada (2009)
5. Bird, C., et al.: Putting It All Together: Using Socio-technical Networks to Predict Failures. In: International Symposium on Software Reliability Engineering (ISSRE), Mysore, India (2009)
6. Williams, L., Kudrjavets, G., Nagappan, N.: On the Effectiveness of Unit Test Automation at Microsoft. In: International Symposium on Software Reliability Engineering (ISSRE), Mysore, India (2010)
7. Nagappan, N., et al.: Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. Empirical Software Engineering (ESE) 13(3), 289–302 (2008)
8. Carver, J.C., Nagappan, N., Page, A.: The Impact of Educational Background on the Effectiveness of Requirements Inspections: An Empirical Study. IEEE Transactions in Software Engineering 34(6), 800–812 (2008)
9. Kudrjavets, G., Nagappan, N., Ball, T.: Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation. In: International Symposium on Soft-ware Reliability Engineering (ISSRE), Raleigh, NC (2006)
10. Conway, M.E.: How Do Committees Invent? Datamation 14(4), 28–31 (1968)
11. Nagappan, N., Murphy, B., Basili, V.: The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In: International Conference on Software Engineering, Leipzig, Germany (2008)

# QuickSpec: Guessing Formal Specifications Using Testing

Koen Claessen[1], Nicholas Smallbone[1], and John Hughes[2]

[1] Chalmers University of Technology
{koen,nicsma}@chalmers.se
[2] Chalmers and Quviq AB
rjmh@chalmers.se

**Abstract.** We present QuickSpec, a tool that automatically generates algebraic specifications for sets of pure functions. The tool is based on testing, rather than static analysis or theorem proving. The main challenge QuickSpec faces is to keep the number of generated equations to a minimum while maintaining completeness. We demonstrate how QuickSpec can improve one's understanding of a program module by exploring the laws that are generated using two case studies: a heap library for Haskell and a fixed-point arithmetic library for Erlang.

## 1 Introduction

Understanding code is hard. But it is vital to understand what code does in order to determine its correctness.

One way to understand code better is to write down one's expectations of the code as formal specifications, which can be tested for compliance, by using a property-based testing tool. Our earlier work on the random testing tool QuickCheck [2] follows this direction. However, coming up with formal specifications is difficult, especially for untrained programmers. Moreover, it is easy to forget to specify certain properties.

In this paper, we aim to aid programmers with this problem. We propose an automatic method that, given a list of function names and their object code, uses testing to come up with a set of *algebraic equations* that seem to hold for those functions. Such a list can be useful in several ways. Firstly, it can serve as a basis for documentation of the code. Secondly, the programmer might gain new insights by discovering new laws about the code. Thirdly, some laws that one expects might be missing (or some laws might be more specific than expected), which points to to a possible miss in the design or implementation of the code.

Since we use testing, our method is potentially unsound, meaning some equations in the list might not hold; the quality of the generated equations is only as good as the quality of the used test data, and care has to be taken. Nonetheless, we still think our method is useful. However, our method is still complete in a precise sense; although there is a limit on the complexity of the expressions that occur in the equations, any syntactically valid equation that actually holds for the code can be derived from the set of equations that QuickSpec generates.

Our method has been implemented for the functional languages Haskell and Erlang in a tool called QuickSpec. At the moment, QuickSpec only works for purely functional code, i.e. no side effects. (Adapting it to imperative and other side-effecting code is ongoing work.)

*Examples.* Let us now show some examples of what QuickSpec can do, by running it on different subsets of the Haskell standard list functions. When we use QuickSpec, we have to specify the functions and variable names which may appear in equations, together with their types. For example, if we generate equations over the list operators

```
(++) :: [Elem] -> [Elem] -> [Elem]      -- list append
(:)  :: Elem -> [Elem] -> [Elem]        -- list cons
[]   :: [Elem]                          -- empty list
```

using variables `x, y, z :: Elem` and `xs, ys, zs :: [Elem]`, then QuickSpec outputs the following list of equations:

```
xs++[] == xs
[]++xs == xs
(xs++ys)++zs == xs++(ys++zs)
(x:xs)++ys == x:(xs++ys)
```

We automatically discover the associativity and unit laws for append (which require induction to prove). These equations happen to comprise a complete characterization of the `++` operator. If we add the list reverse function to the mix, we discover the additional familiar equations

```
reverse [] == []
reverse (reverse xs) == xs
reverse xs++reverse ys == reverse (ys++xs)
reverse (x:[]) == x:[]
```

Again, these laws completely characterize the `reverse` operator. Adding the `sort` function from the standard `List` library, we compute the equations

```
sort [] == []
sort (reverse xs) == sort xs
sort (sort xs) == sort xs
sort (ys++xs) == sort (xs++ys)
sort (x:[]) == x:[]
```

The third equation tells us that `sort` is idempotent, while the second and fourth strongly suggest (but do not imply) that the result of `sort` is independent of the order of its input.

Higher-order functions can be dealt with as well. Adding the function `map` together with a variable `f :: Elem -> Elem`, we obtain:

```
map f [] == []
map f (reverse xs) == reverse (map f xs)
map f xs++map f ys == map f (xs++ys)
f x:map f xs == map f (x:xs)
```

All of the above uses of QuickSpec only took a fraction of a second to run, and the equations shown here is the verbatim output of the tool.

*Main related work.* The existing work that is most similar to ours is [4]. They describe a tool for discovering algebraic specifications from Java classes using testing, using a similar overall approach as ours (there are however important technical differences discussed in the related work section later in the paper). However, the main difference between our work and theirs is that we generate equations between *nested expressions* consisting of functions and variables whereas they generate equations between Java program fragments that are *sequences of method calls.* The main problem we faced when designing the algorithms behind QuickSpec was taming the explosion of equations generated by operators with structural properties, such as associativity and commutativity, equations that are not even expressible as equations between sequences of method calls (results of previous calls cannot be used as arguments to later ones).

*Contributions.* We present a efficient method, based on testing, that automatically computes algebraic equations that seem to hold for a list of specified pure functions. Moreover, using two larger case studies, we show the usefulness of the method, and present concrete techniques of how to use the method effective in order to understand programs better.

## 2 How QuickSpec Works

The input taken by QuickSpec consists of three parts: (a) the compiled program, (b) a list of functions and variables, together with their types, and (c) test data generators for each of the types of which there exists at least one variable. As indicated in the introduction, the quality of the test data determines the quality of the generated equations. As such, it is important to provide good test data generators, that fit the program at hand. In our property-based random testing tool QuickCheck [2], we have a range of test data generators for standard types, and a library of functions for building custom generators.

*The method.* The method used by QuickSpec follows four distinct steps: (1) We first generate a (finite) set of terms, called the *universe*, that includes any term that might occur on either side of an equation. (2) We use testing to partition the universe into *equivalence classes*; any two terms in the same equivalence class are considered equal after the testing phase. (3) We generate a list of *equations* from the equivalence classes. (4) We use *pruning* to filter out equations that follow from other equations by equational reasoning.

In the following, we discuss each of these steps in more detail, plus an optimization that greatly increases the efficiency of the method.

*The universe.* First, we need to pin down what kind of equations QuickSpec should generate. To keep things simple and predictable, we only generate one finite set of terms, the *universe*, and any pair of terms from the universe can form an equation.

How the universe is generated is really up to the user, all we require is that the universe is subterm-closed. The most useful way of generating the universe is letting the user specify a *term depth* (usually 3 or 4), and then simply produce all terms that are not too deep. The terms here consists of the function symbols and variables from the specified API.

The size of the universe is typically 1000 to 50.000 terms, depending on the application.

*Equivalence classes.* The next step is to gather information about the terms in the universe. This is the only step in the algorithm that uses testing, and in fact makes use of the program. Here, we need to determine which terms seem to behave the same, and which terms seem to behave differently. In other words, we are computing an equivalence relation over the terms.

Concretely, in order to compute this equivalence relation, we use a refinement process. We represent the equivalence relation as a set of equivalence classes, partitions of the universe. We start by assuming that all terms are equal, and put all terms in the universe into one giant equivalence class. Then, we repeat the following process: We use the test data generators to generate test data for each of the variables occurring in the terms. We then refine each equivalence class into possibly smaller ones, by evaluating all the terms in a class and grouping together the ones that are still equal, splitting the terms that are different. The process is repeated until the equivalence relation seems "stable"; if no split has happened for the last 200 tests. Note that equivalence classes of size 1 are trivial, and can be discarded; these contain a term that is only equal to itself.

The typical number of non-trivial equivalence classes we get after this process lies between 500 and 5.000, again depending on the size of the original universe and the application.

Once these equivalence classes are generated, the testing phase is over, and the equivalence relation is trusted to be correct for the remainder of the method.

*Equations.* From the equivalence classes, we can generate a list of equations between terms. We do this by picking one representative term $r$ from the class, and producing the equation $t = r$ for all other terms from the class. So, an equivalence class of size $k$ produces $k - 1$ equations. The equations seem to look nicest if $r$ is the *simplest* element of the equivalence class (according to some simplicity measure based on for example depth and/or size), but what $r$ is chosen has no effect on the completeness of the algorithm.

The typical number of equations that are generated in this step lies between 5000 and 50.000. A list of even 5.000 equations is however not something that we want to present to the user; the number of equations should be in the tens, not hundreds or thousands.

There is a great potential for improvement here. Among the equations generated for the boolean operator `&&`, the constant `false`, and the variables `x` and `y` we might find for example:

```
1. x&&false == false  3. y&&false == false  5. false&&false == false
2. false&&x == false  4. false&&y == false  6. x&&y==y&&x
```

It is obvious that law 5 is an instance of laws 1-4, and that laws 3 and 4 are just renamings of laws 1 and 2. Moreover, law 6 and 1 together imply all the other laws. So, the above laws could be replaced by just `x&&false == false` and `x&&y==y&&x`. This is the objective of the pruning step.

*Pruning.* Pruning filters out unnecessary laws. Which laws are kept and which are discarded by our current implementation of QuickSpec is basically an arbitrary choice. What we would like to argue is that it must be an arbitrary choice, but that it should be governed by the following four principles.

(1) *Completeness:* we can only remove a law if it can be derived from the remaining laws. We cannot force the method to remove all such laws, firstly because there is no unique minimal set, and secondly because derivability is not decidable. (2) *Conciseness:* we should however remove all obvious redundancy, for a suitably chosen definition of redundant. (3) *Implementability:* the method should be implementable and reasonably efficient. (4) *Predictability:* the user should be able to draw conclusions from the absence or presence of a particular law, which means that no ad-hoc decisions should be made in the algorithm.

That the choice of what is redundant or not is not obvious became clear to us after long discussions between the authors of the paper on particular example sets of equations. Also, what "derivable" means is not clear either: Do we allow simple equational reasoning, something weaker (because of decidability issues), or something stronger (by adding induction principles)?

Eventually, we settled on the following. First, all equations are ordered according to a simplicity measure. The simplicity measure can again be specified by the user; the default one we provide is based on term size. Then, we go through the list of equations in order, removing any equation that is "derivable" from the previous ones.

For "derivable", we decided to define a decidable and predictable *approximation* of logical implication for equations. The approximation uses a *congruence closure* data-structure, a generalization of a union/find data-structure that maintains a congruence relation[1] over a finite subterm-closed set of terms. Congruence closure is one of the key ingredients in modern SMT-solvers, and we simply reimplemented an efficient modern congruence closure algorithm [8].

Congruence closure enjoys the following property: suppose we have a set of equations $E$, and for each equation $s = t$ we record in the congruence closure data-structure $\equiv$ the fact $s \equiv t$. Then for any terms $a$ and $b$, $a \equiv b$ will be true exactly if $a = b$ can be proved from $E$ using *only* the rules of reflexivity, symmetry, transitivity and congruence of $=$.

This almost gives us a way of checking whether $a = b$ follows from $E$. However, we want to know whether $a = b$ can be proved at all from $E$, not whether it can be proved using some restricted set of rules. There's one more rule we could use in a proof, that's not covered by congruence closure: any instance of a valid equation is valid. To approximate this rule, for each equation $s = t$ in $E$, we should record not just $s \equiv t$ but many *instances* $\sigma(s) \equiv \sigma(t)$ (where $\sigma$ is a substitution).

[1] A congruence relation is an equivalence relation that is also a congruence: if $x \equiv y$ then $C[x] \equiv C[y]$ for all contexts $C$.

In more detail, our algorithm is as follows:

1. We maintain a congruence relation $\equiv$ over terms, which initially is the identity relation. The relation $\equiv$ is going to represent all knowledge implied by accepted equations so far, so that if $s \equiv t$ then $s = t$ is derivable from the accepted equations.
2. We order all equations according to the equation ordering $<$, simplest equations first.
3. We loop through all equations, starting at the simplest. For each equation $s = t$, we check if $s \equiv t$ according to the maintained congruence relation $\equiv$. If so, the equation $s = t$ is implied by previous equations, and we discard it.
4. If $s \not\equiv t$, we produce $s = t$ as an equation. We then update the congruence relation $\equiv$ to represent the fact that we have produced the equation $s = t$. We do this by picking a finite set of instances of $s = t$. That is, we choose a finite set $\Sigma$ of substitutions; then, for each $\sigma \in \Sigma$, we add the fact $\sigma(s) \equiv \sigma(t)$ to the congruence closure data-structure $\equiv$.
5. Once all equations have been taken care of, we are done.

We didn't specify above which instances of each equation $s = t$ to generate. Our original choice was to generate all substitutions $\sigma$ such that $\sigma(s)$ and $\sigma(t)$ were in the universe. This allowed the algorithm to find any proof that only uses terms from the universe.

Now, instead, we generate substitutions separately for $s$ and $t$. In the case of $s$, we generate all substitutions $\sigma$ such that $\sigma(s)$ is in the universe (where $\sigma$ ranges over the variables of $s$) and record $\sigma(s) = \sigma(t)$, and similarly in the case of $t$. This is a relaxation of our earlier choice.

By doing this, we allow the algorithm to reason also about terms $t$ that lie outside the universe, but only if that term $t$ is equated by an equation to a term $s$ that lies inside the universe. This modification does not noticeably influence performance, but allowing this was vital to prune away equations involving operators with structural properties, such as commutativity and associativity. For example, generating properties about the arithmetic operator `+`, only allowing reasoning within the universe, we end up with:

```
1. x+y = y+x
2. y+(x+z) = (z+y)+x
3. (x+y)+(x+z) = (z+y)+(x+x)
```

The third equation can be derived from the first two, but we need to use a term `x+(y+(x+z))` that lies outside of the universe. Adding the modification we just described to the algorithm, this last equation is also pruned away.

*The depth optimisation.* QuickSpec includes one optimisation to reduce the number of terms generated. We will first motivate the optimisation and then explain it in more detail.

Suppose we have run QuickSpec on an API of boolean operators with a depth limit of 2, giving (among others) the law `x&&x==x`. But now, suppose we want to increase the depth limit on terms from 2 to 3. Using the algorithm described above, we would first generate all terms of depth 3, including such ones as `x&&y` and `(x&&x)&&y`. But these two terms are obviously equivalent (since we know that `x&&x==x`), we won't get any more laws by generating both of them, and we ought to generate only `x&&y` and not `(x&&x)&&y`.

The observation we make is that, if two terms are equal (like `x&&x` and `x` above), we ought to pick one of them as the "canonical form" of that expression; we avoid generating any term that has a non-canonical form as a subterm. In this example, we don't generate `(x&&x)&&y`, because it has `x&&x` as a subterm. (We do still generate `x&&x` on its own, otherwise we wouldn't get the law `x&&x==x`.)

The depth optimisation applies this observation, and works quite straightforwardly. If we want to generate all terms up to depth 3, say, we first generate all terms up to depth 2 (recursively applying the same optimisation) and sort them into equivalence classes by testing. Then we only generate those terms for which all the direct subterms are the representative of their equivalence class.

We can justify why this optimisation is sound. If we choose not to generate a term `t` with canonical form `t'`, and there would have been an equation `t==u`, there will also be an equation `t'==u`.[2] Furthermore, we will record in the congruence closure data structure all of the equations necessary to prove `t==t'` (all of these terms in these equations have smaller depth than `t`) so our pruning algorithm would have been able to prove `t==u` anyway.)

This optimisation makes a very noticeable difference to the number of terms generated. For a large list signature, the number of terms goes down from 21266 to 7079. For booleans there is a much bigger difference, since so many terms are equal: without the depth optimisation we generate 7395 terms, and with it 449 terms. Time-wise, the method becomes an order of a magnitude faster.

## 3 Case Studies

In this section, we present two case studies using QuickSpec. Our goal is primarily to derive *understanding* of the code we test. In many cases, the specifications generated by QuickSpec are initially disappointing—but by *extending the signature with new operations* we are able to arrive at concise and perspicuous specifications. Arguably, selecting the right operations to specify is a key step in formulating a good specification, and one way to see QuickSpec is as a tool to support exploration of this design space.

### 3.1 Case Study #1: Leftist Heaps in Haskell

A *leftist heap* [9] is a data structure that implements a priority queue. A leftist heap provides the usual heap operations:

```
empty :: Heap                      findMin :: Heap -> Elem
isEmpty :: Heap -> Bool            deleteMin :: Heap -> Heap
insert :: Elem -> Heap -> Heap
```

[2] This relies on `t'` not having greater depth than `t`, which requires the term ordering to always pick the representative of an equivalence class as a term with the smallest depth.

When we tested this signature with the variables `h, h1, h2 :: Heap` and `x, y, z :: Elem`. then QuickSpec generated a rather incomplete specification. The specification describes the behaviour of `findMin` and `deleteMin` on empty and singleton heaps:[3]

```
findMin empty == undefined          deleteMin empty == undefined
findMin (insert x empty) == x       deleteMin (insert x empty) == empty,
```

It shows that the order of insertion into a heap is irrelevant:

```
insert y (insert x h) == insert x (insert y h),
```

Apart from that, it only contains the following equation:

```
isEmpty (insert x h1) == isEmpty (insert x h)
```

This last equation is quite revealing—obviously, we would expect both sides to be `False`, which explains why they are equal. But why doesn't QuickSpec simply print the equation `isEmpty (insert x h) == False`? The reason is that `False` is not in our signature! When we add it to the signature, then we do indeed obtain the simpler form instead of the original equation above.[4]

In general, when a term is found to be equal to a renaming of itself with different variables, then this is an indication that a constant should be added to the signature, and in fact QuickSpec prints a suggestion to do so.

Generalising a bit, since `isEmpty` returns a `Bool`, it's certainly sensible to give QuickSpec operations that manipulate booleans. We added the remaining boolean connectives `True`, `&&`, `||` and `not`, and one newly-expressible law appeared, `isEmpty empty == True`.

**Merge.** Leftist heaps actually provide one more operation than those we encountered so far: merging two heaps.

```
merge :: Heap -> Heap -> Heap
```

If we run QuickSpec on the new signature, we get the fact that `merge` is commutative and associative and has `empty` as a unit element:

```
merge h1 h == merge h h1
merge h1 (merge h h2) == merge h (merge h1 h2)
merge h empty == h
```

We get nice laws about `merge`'s relationship with the other operators:

```
merge h (insert x h1) == insert x (merge h h1)
isEmpty h && isEmpty h1 == isEmpty (merge h h1)
```

We also get some curious laws about merging a heap with itself:

```
findMin (merge h h) == findMin h
merge h (deleteMin h) == deleteMin (merge h h)
```

[3] QuickSpec generates an exceptional value `undefined` at each type.

[4] For completeness, we will list all of the new laws that QuickSpec produces every time we change the signature.

These are *all* the equations that are printed. Note that there are no redundant laws here. As mentioned earlier, our testing method guarantees that this set of laws is *complete*, in the sense that any valid equation over our signature, which is not excluded by the depth limit, follows from these laws.

**With Lists.** We can get useful laws about heaps by relating them to a more common data structure, *lists*. First, we need to extend the signature with operations that convert between heaps and lists:

```
fromList :: [Elem] -> Heap
toList :: Heap -> [Elem]
```

`fromList` turns a list into a heap by folding over it with the `insert` function; `toList` does the reverse, deconstructing a heap using `findMin` and `deleteMin`. We should also add a few list operations mentioned earlier: `++`, `tail`, `:`, `[]`, and `sort`; and variables `xs, ys, zs :: [Elem]`. Now, QuickSpec discovers many new laws. The most striking one is

```
toList (fromList xs) == sort xs.
```

This is the definition of heapsort! The other laws indicate that our definitions of `toList` and `fromList` are sensible:

```
sort (toList h) == toList h
fromList (toList h) == h
fromList (sort xs) == fromList xs
fromList (ys++xs) == fromList (xs++ys)
```

The first law says that `toList` produces a sorted list, and the second that `fromList . toList` is the identity (up to `==` on heaps, which actually applies `toList` to each operand and compares them!). The other two laws suggest that the order of `fromList`'s input doesn't matter.

We get a definition by pattern-matching of `fromList`:

```
fromList [] == empty
insert x (fromList xs) == fromList (x:xs)
merge (fromList xs) (fromList ys) == fromList (xs++ys)
```

We also get a family of laws relating heap operations to list operations:

```
toList empty == []
head (toList h) == findMin h
toList (deleteMin h) == tail (toList h)
```

We can think of `toList h` as an abstract model of `h`—all we need to know about a heap is the sorted list of elements, in order to predict the result of any operation on that heap. The heap itself is just a clever representation of that sorted list of elements.

The three laws above define `empty`, `findMin` and `deleteMin` by how they act on the sorted list of elements—the model of the heap. For example, the third law says that applying `deleteMin` to a heap corresponds to taking the `tail` in the abstract model (a sorted list). Since `tail` is obviously the correct way to remove the minimum element from a sorted list, this equation says exactly that `deleteMin` is correct![5]

So these three equations are a complete specification of `empty`, `findMin` and `deleteMin`!

This section highlights the importance of choosing a rich set of operators when using QuickSpec. There are often useful laws about a library that mention functions from unrelated libraries; the more such functions we include, the more laws QuickSpec can find. In the end, we got a complete specification of heaps (and heapsort, as a bonus!) by including list functions in our testing.

It's not always obvious *which* functions to add to get better laws. In this case, there are several reasons for choosing lists: they're well-understood, there are operators that convert heaps to and from lists, and sorted lists form a model of priority queues.

**Buggy Code.** What happens when the code under test has a bug? To find out, we introduced a fault into `toList`. The buggy version of `toList` doesn't produce a sorted list, but rather the elements of the heap in an arbitrary order.

We were hoping that some laws would fail, and that QuickSpec would produce *specific instances* of some of those laws instead. This happened: whereas before, we had many useful laws about `toList`, afterwards, we had only two:

```
toList empty == []
toList (insert x empty) == x:[]
```

Two things stand out about this set of laws: first, the law `sort (toList h) == toList h` does not appear, so we know that the buggy `toList` doesn't produce a sorted result. Second, we *only get equations about empty and singleton heaps*, not about heaps of arbitrary size. QuickSpec is unable to find *any* specification of `toList` on nontrivial heaps, which suggests that the buggy `toList` *has* no simple specification.

## 3.2 Case Study #2: Understanding a Fixed Point Arithmetic Library in Erlang

We used QuickSpec to try to understand a library for fixed point arithmetic, developed by a South African company, which we were previously unfamiliar with. The library exports 16 functions, which is rather overwhelming to analyze in one go, so we decided to generate equations for a number of different subsets of the API instead. In this section, we give a detailed account of our experiments and developing understanding.

Before we could begin to use QuickSpec, we needed a QuickCheck generator for fixed point data. We chose to use one of the library functions to ensure a valid result, choosing one which seemed able to return arbitrary fixed point values:

```
fp() -> ?LET({N,D},{largeint(),nat()},from_minor_int(N,D)).
```

[5] This style of specification is not new and goes back to Hoare [5].

That is, we call `from_minor_int` with random arguments. We suspected that `D` is the precision of the result—a suspicion that proved to be correct.

**Addition and Subtraction.** We began by testing the `add` operation, deriving commutativity and associativity laws as expected. Expecting laws involving zero, we defined

```
zero() -> from_int(0)
```

and added it to the signature, obtaining as our reward a unit law, `add(A,zero()) == A`.

The next step was to add subtraction to the signature. However, this led to several very similar laws being generated—for example,

```
add(B,add(A,C)) == add(A,add(B,C))
add(B,sub(A,C)) == add(A,sub(B,C))
sub(A,sub(B,C)) == add(A,sub(C,B))
sub(sub(A,B),C) == sub(A,add(B,C))
```

To relieve the problem, we added another derived operator to the signature instead:

```
negate(A) -> sub(zero(),A).
```

and observed that the earlier family of similar laws was no longer generated, replaced by a single one, `add(A,negate(B)) == sub(A,B)`. Thus by *adding* a new auxiliary function to the signature, `negate`, we were able to *reduce* the complexity of the specification considerably.

After this new equation was generated by QuickSpec, we tested it extensively using *QuickCheck*. Once confident that it held, we could safely *replace* `sub` in our signature by `add` and `negate`, without losing any other equations. Once we did this, we obtained a more useful set of new equations:

```
add(negate(A),add(A,A)) == A
add(negate(A),negate(B)) == negate(add(A,B))
negate(negate(A)) == A
negate(zero()) == zero()
```

These are all very plausible—what is striking is the *absence* of the following equation:

```
add(A,negate(A)) == zero()
```

When an expected equation like this is missing, it is easy to formulate it as a QuickCheck property and find a counterexample, in this case `{fp,1,0,0}`. We discovered by experiment that `negate({fp,1,0,0})` is actually the same value! This strongly suggests that this is an alternative representation of zero (`zero()` evaluates to `{fp,0,0,0}` instead).

**$0 \neq 0$.** It is reasonable that a fixed point arithmetic library should have different representations for zero of different precisions, but we had not anticipated this. Moreover, since we want to derive equations involving zero, the question arises of *which zero* we would like our equations to contain! Taking our cue from the missing equation, we introduced a new operator `zero_like(A) -> sub(A,A)` and then derived not only `add(A,negate(A)) == zero_like(A)` but a variety of other interesting laws. These two equations suggest that the result of `zero_like` depends only on the number of decimals in its argument,

```
zero_like(from_int(I)) == zero()
zero_like(from_minor_int(J,M)) == zero_like(from_minor_int(I,M))
```

this equation suggests that the result has the *same* number of decimals as the argument,

```
zero_like(zero_like(A)) == zero_like(A)
```

while these two suggest that the number of decimals is preserved by arithmetic.

```
zero_like(add(A,A)) == zero_like(A)
zero_like(negate(A)) == zero_like(A)
```

It is not in general true that `add(A,zero_like(B)) == A` which is not so surprising—the precision of `B` affects the precision of the result. QuickSpec does find the more restricted property, `add(A,zero_like(A)) == A`.

**Multiplication and Division.** When we added multiplication and division operators to the signature, then we followed a similar path, and were led to introduce `reciprocal` and `one_like` functions, for similar reasons to `negate` and `zero_like` above. One interesting equation we discovered was this one:

```
divide(one_like(A),reciprocal(A)) == reciprocal(reciprocal(A))
```

The equation is clearly true, but why is the right hand side `reciprocal(reciprocal(A))`, instead of just `A`? The reason is that the left hand side raises an exception if `A` is zero, and so the right hand side must do so also—which `reciprocal(reciprocal(A))` does.

We obtain many equations that express things about the precision of results, such as

```
multiply(B,zero_like(A)) == zero_like(multiply(A,B))
multiply(from_minor_int(I,N),from_minor_int(J,M)) ==
           multiply(from_minor_int(I,M),from_minor_int(J,N))
```

where the former expresses the fact that the precision of the zero produced depends both on `A` and `B`, and the latter expresses

$$i \times 10^{-m} \times j \times 10^{-n} = i \times 10^{-n} \times j \times 10^{-m}$$

That is, it is in a sense the commutativity of multiplication in disguise.

One equation we expected, but did *not* see, was the distributivity of multiplication over addition. Alerted by its absence, we formulated a corresponding QuickCheck property,

```
prop_multiply_distributes_over_add() ->
    ?FORALL({A,B,C},{fp(),fp(),fp()},
        multiply(A,add(B,C)) == add(multiply(A,B),multiply(A,C))).
```

and used it to find a counterexample:

```
A = {fp,1,0,4}, B = {fp,1,0,2}, C = {fp,1,1,4}
```

We used the library's `format` function to convert these to strings, and found thus that $A = 0.4, B = 0.2, C = 1.4$. Working through the example, we found that multiplying `A` and `B` returns a representation of 0.1, and so we were alerted to the fact that `multiply` rounds its result to the precision of its arguments.

**Understanding Precision.** At this point, we decided that we needed to understand how the precision of results was determined, so we defined a function `precision` to extract the first component of an `{fp,...}` structure, where we suspected the precision was stored. We introduced a `max` function on naturals, guessing that it might be relevant, and (after observing the term `precision(zero())` in generated equations) the constant natural zero. QuickSpec then generated equations that tell us rather precisely how the precision is determined, including the following:

```
max(precision(A),precision(B)) == precision(add(A,B))
precision(divide(zero(),A)) == precision(one_like(A))
precision(from_int(I)) == 0
precision(from_minor_int(I,M)) == M
precision(multiply(A,B)) == precision(add(A,B))
precision(reciprocal(A)) == precision(one_like(A))
```

The first equation tells us the addition uses the precision of whichever argument has the most precision, and the fifth equation tells us that multiplication does the same. The second and third equations confirm that we have understood the representation of precision correctly. The second and sixth equations reveal that our definition of `one_like(A)` raises an exception when `A` is zero—this is why we do not see `precision(one_like(A)) == precision(A)`.

The second equation is more specific than we might expect, and in fact it is true that

```
precision(divide(A,B)) == max(precision(A),precision(one_like(B)))
```

but the right hand side exceeds our depth limit, so QuickSpec cannot discover it.

**Adjusting Precision.** The library contained an operation whose meaning we could not really guess from its name, `adjust`. Adding `adjust` to the signature generated a set of equations including the following:

```
adjust(A,precision(A)) == A
precision(adjust(A,M)) == M
zero_like(adjust(A,M)) == adjust(zero(),M)
adjust(zero_like(A),M) == adjust(zero(),M)
```

These equations make it fairly clear that `adjust` sets the precision of its argument. We also generated an equation relating double to single adjustment:

```
adjust(adjust(A,M),0) == adjust(A,0)
```

**Summing Up.** Overall, we found QuickSpec to be a very useful aid in developing an understanding of the fixed point library. Of course, we could simply have formulated the expected equations as QuickCheck properties, and tested them without the aid of QuickSpec. However, this would have taken very much longer, and because the work is fairly tedious, there is a risk that we might have forgotten to include some important properties. QuickSpec automates the tedious part, and allowed us to spot missing equations quickly.

Of course, QuickSpec also generates *unexpected* equations, and these would be much harder to find using QuickCheck. In particular, when investigating functions such as `adjust`, where we initially had little idea of what they were intended to do, then it would have been very difficult to formulate candidate QuickCheck properties in advance.

We occasionally encountered false equations resulting from the unsoundness of the method. In some cases these showed us that we needed to improve the distribution of our test data, in others (such as the difference between rounding in two stages and one stage) then the counterexamples are simply hard to find. QuickSpec runs relatively few tests of each equation (a few hundred), and so, once the most interesting equations have been selected, then it is valuable to QuickCheck them many more times.

## 4 Related Work

As mentioned earlier, the existing work that is most similar to ours is [4]; a tool for discovering algebraic specifications from Java classes. They generate terms and evaluate them, dynamically identify terms which are equal, then generate equations and filter away redundant ones. There are differences in the kind of equations that can be generated, which have been discussed earlier. The most important difference with our method is the fact that they initially generate only *ground* terms when searching for equations, then later generalise the ground equations by introducing variables, and test the equations *using the ground terms as test data*. To get good test data, then, they need to generate a large set of terms to work with, which heavily effects the efficiency of the subsequent generalization and pruning phases. In contrast, in our system, the number of terms does not affect the quality of the test data. So we get away with generating fewer terms—the cost of generating varying test data is only paid during testing, i.e. during the generation of the equivalence relation, and not in the term generation or pruning phase. Furthermore, we don't need a generalisation phase because our terms contain variables from the start.

There are other differences as well. They use a heuristic term-rewriting method for pruning equations; we use a predictable congruence closure algorithm. They observe—as we do—that conditional equations would be useful, but neither tool generates them. Our tool appears to be faster (our examples take seconds to run, while comparable examples in their setting take hours). It is unfortunately rather difficult to make a fair comparison between the efficacy and performance of the two approaches, because their tool and examples are not available for download.

*Daikon* is a tool for inferring likely invariants in C, C++, Java or Perl programs [3]. Daikon observes program variables at selected program points during testing, and applies machine learning techniques to discover relationships between them. For example, Daikon can discover linear relationships between integer variables, such as array indices. Agitar's commercial tool based on Daikon generates test cases for the code under analysis automatically [1]. However, Daikon will not discover, for example, that `reverse(reverse(Xs)) == Xs`, unless such a double application of `reverse` appears in the program under analysis. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms constructed using an API. This is analogous to the difference between *assertions* placed in program code, and the kind of *properties* which QuickCheck tests, that also invoke the API under test in interesting ways. While Daikon's approach is ideal for imperative code, especially code which loops over arrays, QuickSpec is perhaps more appropriate for analysing pure functions.

*Inductive logic programming* (ILP) [7] aims to infer logic programs from examples—specific instances—of their behaviour. The user provides both a collection of true statements and a collection of false statements, and the ILP tool finds a program consistent with those statements. Our approach only uses *false* statements as input (inequality is established by testing), and is optimized for deriving equalities.

In the area of *Automated Theorem Discovery* (ATD), the aim is to emulate the human theorem discovery process. The idea can be applied to many different fields, such as mathematics, physics, but also formal verification. An example of an ATD system for mathematicians is MathSaid [6]. The system starts by generating a finite set of *hypotheses*, according to some syntactical rules that capture typical mathematical thinking, for example: if we know $A \Rightarrow B$, we should also check if $B \Rightarrow A$, and if not, under what conditions this holds. Theorem proving techniques are used to select theorems and patch non-theorems. Since this leads to many theorems, a filtering phase decides if theorems are interesting or not, according to a number of different predefined "tests". One such test is the simplicity test, which compares theorems for simplicity based on their proofs, and only keeps the simplest theorems. The aim of their filtering is quite different from ours (they want to filter out theorems that mathematicians would have considered trivial), but the motivation is the same; there are too many theorems to consider.

## 5 Conclusions and Future Work

We have presented a new tool, QuickSpec, which can automatically generate algebraic specifications for functional programs. Although simple, it is remarkably powerful. It can be used to aid program understanding, or to generate a QuickCheck test suite to detect changes in specification as the code under test evolves. We are hopeful that it will enable more users to overcome the barrier that formulating properties can present, and discover the benefits of QuickCheck-style specification and testing.

For future work, we plan to generate conditional equations. In some sense, these can be encoded in what we already have by specifying new custom types with appropriate operators. For example, if we want `x<=y` to occur as a precondition, we might introduce a type `AscPair` of "pairs with ascending elements", and add the functions `smaller,larger :: AscPair -> Int` and the variable `p :: AscPair` to the API. A conditional equation we could then generate is:

```
isSorted (smaller p : larger p : xs) == isSorted (larger p : xs)
```

(Instead of the perhaps more readable `x<=y ==> isSorted (x:y:xs) == isSorted (y:xs)`.) But we are still investigating the limitations and applicability of this approach.

Another class of equations we are looking at are equations between program fragments that can have side effects.

## References

1. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 169–180. ACM, New York (2006)
2. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP 2000: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp. 268–279. ACM, New York (2000)
3. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
4. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. IEEE Trans. Software Eng. 33(8), 526–543 (2007)
5. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inf. 1, 271–281 (1972)
6. McCasland, R.L., Bundy, A.: Mathsaid: a mathematical theorem discovery tool. In: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2006 (2006)
7. Muggleton, S., de Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19, 629–679 (1994)
8. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)
9. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)

# Testing First-Order Logic Axioms in Program Verification

Ki Yung Ahn[1,2] and Ewen Denney[3]

[1] Portland State University, Portland, OR 97207-0751, USA
[2,⋆] Mission Critical Technologies, Inc. / NASA Ames Research Center, Moffett Field, CA 94035, USA
[3] Stinger Ghaffarian Technologies, Inc. / NASA Ames Research Center, Moffett Field, CA 94035, USA
kya@cs.pdx.edu, Ewen.W.Denney@nasa.gov

**Abstract.** Program verification systems based on automated theorem provers rely on user-provided axioms in order to verify domain-specific properties of code. However, formulating axioms correctly (that is, formalizing properties of an intended mathematical interpretation) is non-trivial in practice, and avoiding or even detecting unsoundness can sometimes be difficult to achieve. Moreover, speculating soundness of axioms based on the output of the provers themselves is not easy since they do not typically give counterexamples. We adopt the idea of model-based testing to aid axiom authors in discovering errors in axiomatizations. To test the validity of axioms, users define a computational model of the axiomatized logic by giving interpretations to the function symbols and constants in a simple declarative programming language. We have developed an axiom testing framework that helps automate model definition and test generation using off-the-shelf tools for meta-programming, property-based random testing, and constraint solving. We have experimented with our tool to test the axioms used in AutoCert, a program verification system that has been applied to verify aerospace flight code using a first-order axiomatization of navigational concepts, and were able to find counterexamples for a number of axioms.

**Keywords:** model-based testing, program verification, automated theorem proving, property-based testing, constraint solving.

## 1 Introduction

### 1.1 Background

Program verification systems based on automated theorem provers rely on user-provided axioms in order to verify domain-specific properties of code. AutoCert [1] is a source code verification tool for autogenerated code in safety critical domains, such as flight code generated from Simulink models in the guidance, navigation, and control (GN&C) domain using MathWorks' Real-Time Workshop

⋆ MCT/NASA Ames internship program.

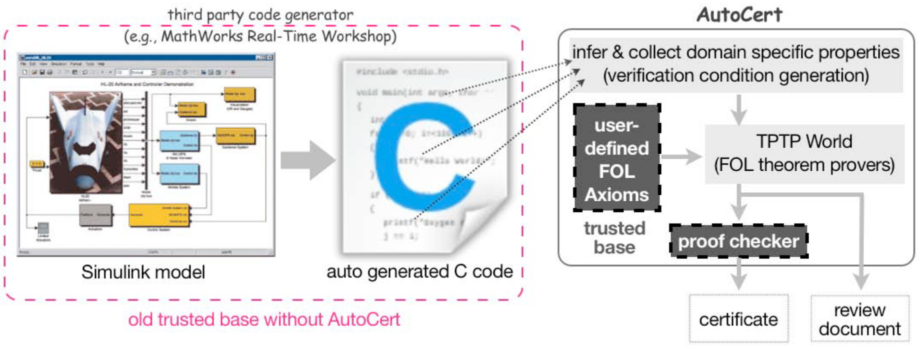

third party code generator
(e.g., MathWorks Real-Time Workshop)
Simulink model
auto generated C code
old trusted base without AutoCert
AutoCert
infer & collect domain specific properties (verification condition generation)
user-defined FOL Axioms
trusted base
TPTP World (FOL theorem provers)
proof checker
certificate
review document

**Fig. 1.** AutoCert narrows down the trusted base by verifying the generated code

code generator. AutoCert supports certification by formally verifying that the generated code complies with a range of mathematically specified requirements and is free of certain safety violations. AutoCert uses Automated Theorem Provers (ATPs) [2] based on First-Order Logic (FOL) to formally verify safety and functional correctness properties of autogenerated code, as illustrated in Figure 1.

AutoCert works by inferring logical annotations on the source code, and then using a verification condition generator (VCG) to check these annotations. This results in a set of first-order verification conditions (VCs) that are then sent to a suite of ATPs. These ATPs try to build proofs based on the user-provided axioms, which can themselves be arbitrary First-Order Formulas (FOFs).

If all the VCs are successfully proven, then it is guaranteed that the code complies with the properties[1] – with one important proviso: we need to trust the verification system, itself. The *trusted base* is the collection of components which must be correct for us to conclude that the code itself really is correct. Indeed, one of the main motivations for applying a verification tool like AutoCert to autocode is to remove the code generator—a large, complex, black box—from the trusted base.

The annotation inference system is not part of the trusted base, since annotations merely serve as hints (albeit necessary ones) in the verification process—they are ultimately checked via their translation into VCs by the VCG. The logic that is encoded in the VCG does need to be trusted but this is a relatively small and stable part of the system. The ATPs do not need to be trusted since the proofs they generate can (at least, in principle) be sent to a proof checker [3]. In fact, it is the domain theory, defined as a set of logical axioms, that is the most crucial part of the trusted base. Moreover, in our experience, it is the most common source of bugs.

However, formulating axioms correctly (i.e., precisely as the domain expert really intends) is non-trivial in practice. By correct we mean that the axioms

[1] The converse is not always true, however: provers can time out or the domain theory might be incomplete.

formulate properties of an intended mathematical interpretation. The challenges of axiomatization arise from several dimensions. First, the domain knowledge has its own complexity. AutoCert has been used to verify mathematical requirements on navigation software that carries out various geometric coordinate transformations involving matrices and quaternions. Axiomatic theories for such constructs are complex enough that mistakes are not uncommon. Second, the axioms frequently need to be modified in order to formulate them in a form suitable for use with ATPs. Such modifications tend to obscure the axioms further. Third, it is easy to accidentally introduce unsound axioms due to implicit, but often incompatible interpretations of the axioms. Fourth, speculating on the validity of axioms from the output of existing ATPs is difficult since theorem provers typically do not give any examples or counterexamples (and some, for that matter, do not even give proofs).

### 1.2 Overview

We adopt the idea of model-based testing to aid axiom authors and domain experts in discovering errors in axiomatization. To test the validity of axioms, users define a computational model of the axiomatized logic by giving interpretations to each of the function symbols and constants as computable functions and data constants in a simple declarative programming language. Then, users can test axioms against the computational model with widely used software testing tools. The advantage of this approach is that the users have a concrete intuitive model with which to test validity of the axioms, and can observe counterexamples when the model does not satisfy the axioms.

In §2 we develop a sequence of simple axioms for rotation matrices, and use these to motivate this work by showing some of the pitfalls in developing axioms. Then §3 shows some examples of axioms used by AutoCert and the issues that arise in testing them. §4 describes the implementation and evaluation of our testing framework. We conclude with a discussion of related work (§5) and thoughts for future work (§6).

## 2 Axioms for Program Verification

In vehicle navigation software, *frames of reference* are used to represent different coordinate systems within which the position and orientation of objects are measured. Navigation software frequently needs to translate between different frames of reference, such as between vehicle-based and earth-based frames when communicating between mission control and a spacecraft. A transformation between two different frames can be represented by a so-called direction cosine matrix (DCM) [4,5]. Verifying navigation software therefore requires us to check that the code is correctly carrying out these transformations, that is, correctly represents these matrices, quaternions, and the associated transformations. As we will show, however, axiomatizing these definitions and their properties is error-prone.

In the following subsections we will use a simplified running example of a two-dimensional rotation matrix (rather than a 3D transformation matrix).

## 2.1 Axiomatizing a Two-Dimensional Rotation Matrix

The two dimensional rotation matrix for an angle $T$ is given by $\begin{pmatrix} \cos(T) & \sin(T) \\ -\sin(T) & \cos(T) \end{pmatrix}$. Matrices in control code are usually implemented using arrays, and the most obvious way to axiomatize these arrays in FOL is extensionally, i.e.,

$$\texttt{select}(A, 0) = \texttt{cos}(T) \wedge \texttt{select}(A, 1) = \texttt{sin}(T) \wedge \cdots$$

but this is unlikely to prove useful in practice. Consider the C implementation `init` in Table 1, which is intended to initialize a two dimensional rotation matrix. A VCG will apply the usual array update rules to derive that the output `X` should be replaced by `update(update(update(update(a, 0, cos(t) ), 1, sin(t)) , 2, uminus(sin(t))), 3, cos(t))`. Unfortunately, provers are generally unable to relate this update term to the extensional definition so, instead, we use the following axiom, written in TPTP first-order formula (FOF) syntax, which defines an array representation of the two-dimensional rotation matrix as a binary relation `rot2D` between an array `A` and angle `T`.

```
fof(rotation2D_def, axiom, ![A,T]:( (lo(A)=0 & hi(A)=3)
                              => rot2D(update(update(update(update(A
                                     , 0,       cos(T) ), 1,sin(T))
                                     , 2,uminus(sin(T))), 3,cos(T)), T) ) ).
```

The function `init` can be specified with precondition `(lo(a)=0&hi(a)=3)` and postcondition `rot2D(X,t)`, where `X` is the function output. In practice, we also have conditions on the physical types of variables (e.g., that $T$ is an angle), but omit this here. Using this specification for `init` gives the verification condition `vc` in Table 1. We can prove `vc` from the axiom `rotation2D_def` alone using two provers from SystemOnTPTP [2], as shown in the first row of Table 2. We chose EP 1.1 and Equinox 4.1 here because these two provers use different strategies. In general, it is necessary to use a combination of provers in order to prove all the VCs arising in practice.

## 2.2 Adding More Axioms

Initialization routines often perform additional operations that do not affect the initialization task. For example, `init1` and `init2` in Table 1 assign some other values to the array elements before initializing them to the values of the rotation matrix elements. Although there are some extra operations, both `init1` and `init2` are, in fact, valid definitions of rotation matrices since they both finally overwrite the array elements to the same values as in `init`. However, we cannot prove the verification conditions generated from these functions from the axiom `rotation2D_def` alone (Table 2), because the theorem provers do not know that *two consecutive updates on the same index are the same as one latter update.*

We can formalize this as the following axiom:

```
fof(update_last, axiom,
 ![A,I,X,Y] : update(update(A,I,X),I,Y) = update(A,I,Y) ).
```

**Table 1.** 2D rotation matrix code and corresponding verification conditions

| C code | Verification Condition |
|---|---|
| `void init(float a[], float t)`<br>`{`<br>` a[0]= cos(t); a[1]= sin(t);`<br>` a[2]=-sin(t); a[3]= cos(t);`<br>`}` | `fof(vc, conjecture, ((lo(a)=0 & hi(a)=3)`<br>`      => rot2D(update(update(`<br>`               update(update(a`<br>`          ,0,       cos(t) ),1,sin(t))`<br>`          ,2,uminus(sin(t))),3,cos(t))`<br>`         ,t))).` |
| `void init1(float a[], float t)`<br>`{`<br>` a[0]= sin(t);`<br>` a[0]= cos(t); a[1]= sin(t);`<br>` a[2]=-sin(t); a[3]= cos(t);`<br>`}` | `fof(vc1, conjecture, ((lo(a)=0 & hi(a)=3)`<br>`      => rot2D(update(`<br>`               update(update(`<br>`               update(update(a`<br>`          ,0,       sin(t) )`<br>`          ,0,       cos(t) ),1,sin(t))`<br>`          ,2,uminus(sin(t))),3,cos(t))`<br>`         ,t))).` |
| `void init2(float a[], float t)`<br>`{`<br>` a[0]= sin(t); a[1]= sin(t);`<br>` a[2]= sin(t); a[3]= sin(t);`<br>` a[0]= cos(t); a[1]= sin(t);`<br>` a[2]=-sin(t); a[3]= cos(t);`<br>`}` | `fof(vc2, conjecture, ((lo(a)=0 & hi(a)=3)`<br>`      => rot2D(update(update(`<br>`               update(update(`<br>`               update(update(`<br>`               update(update(a`<br>`          ,0,       sin(t) ),1,sin(t))`<br>`          ,2,       sin(t) ),3,sin(t))`<br>`          ,0,       cos(t) ),1,sin(t))`<br>`          ,2,uminus(sin(t))),3,cos(t))`<br>`         ,t))).` |
| `void initX(float a[], float t)`<br>`{`<br>` a[0]=-cos(t); a[1]= sin(t);`<br>` a[2]=-sin(t); a[3]= cos(t);`<br>`}` | `fof(vcX, conjecture, ((lo(a)=0 & hi(a)=3)`<br>`      => rot2D(update(update(`<br>`               update(update(a`<br>`          ,0,uminus(cos(t))),1,sin(t))`<br>`          ,2,uminus(sin(t))),3,cos(t))`<br>`         ,t))).` |

**Table 2.** Results of running EP and Equinox through the SystemOnTPTP website with default settings and a timeout of 60 seconds

| VC | axioms | EP (eprover) 1.1 | Equinox 4.1 |
|---|---|---|---|
| `vc` | `rotation2D_def` | Theorem | Theorem |
| `vc1` | `rotation2D_def` | CounterSatisfiable | Timeout |
| | `rotation2D_def`, `update_last` | Theorem | Theorem |
| `vc2` | `rotation2D_def`, `update_last` | CounterSatisfiable | Timeout |
| | `rotation2D_def`, `update_last`, `update_commute` | Theorem | Timeout |
| `vcX` | `rotation2D_def` | CounterSatisfiable | Timeout |
| | `rotation2D_def`, `update_last` | CounterSatisfiable | Timeout |
| | `rotation2D_def`, `update_last`, `update_commute` | Theorem | Theorem |

Then, both EP and Equinox can prove `vc1` as a theorem from the two axioms `rotation2D_def` and `update_last`, as shown in Table 2.

The verification condition `vc2` generated from `init2` is not provable even with the `update_last` axiom added. This is because `init2` has more auxiliary array updates before matrix initialization, and `update_last` axiom is not applicable since none of the consecutive updates are on the same index. To prove that `init2` is indeed a valid initialization routine we need the property that *two consecutive independent updates can switch their order*. The following axiom tries to formalize this property.

```
fof(update_commute, axiom,
 ![A,I,J,X,Y] : update(update(A,I,X),J,Y) = update(update(A,J,Y),I,X) ).
```

With this axiom added, EP can prove `vc2` from the three axioms `rotation2D_def`, `update_last`, and `update_commute`, but strangely, Equinox times out. It is true that some theorem provers can quickly find proofs while others are lost, depending on the conjecture. Nevertheless, considering the simplicity of the formulae, the timeout of Equinox seems quite strange and might indicate a problem.

## 2.3 Detecting Unsoundness and Debugging Axioms

It is important to bear in mind when adding new axioms that we are always at risk of introducing unsoundness. One way to detect unsoundness is to try proving obviously invalid conjectures.[2] For example, the verification condition `vcX` for the incorrect initialization routine `initX` is invalid. The function `initX` is an incorrect implementation of the rotation matrix (§2.1) because `-cos(t)` is assigned to the element at index 0 instead of `cos(t)`, and hence does not satisfy `rot2D`. However, both EP and Equinox can prove `vcX`. The problem is that we have not thoroughly formalized the property that *independent* updates commute in the axiom `update_commute` (see §3.2).

Note that the theorem provers have not guided us to the suspicious axiom as the source of unsoundness. We decided to examine the axioms based on our own experience and insights, not just because Equinox timed out. Theorem provers may also time out while trying to prove valid conjectures from sound axioms. We should not expect that the most recently added axiom is always the cause of unsoundness. Coming up with an invalid conjecture that can be proven, and thus shows that the axioms are unsound, is usually an iterative process. We used our own intuition to find the cause of the problem, again with no help from the provers. Finally, note that the axiom `rotation2D_def` is already quite different from the natural definition of the matrix given above.

In this section, we have shown that it can be difficult to debug unsoundness of the axioms used in program verification systems even for three simple axioms. In practice, we need to deal with far larger sets of axioms combining multiple theories. In the following section, we will show how our method of testing axioms

[2] It is not enough to just try and prove false since different provers exploit inconsistency in different ways. Moreover, a logic can be consistent yet still be unsound with respect to a model.

against a computational model helps us to detect problems in axioms more easily and systematically.

## 3 Testing Axioms

When we have a computational model, we can run tests on logical formulae against that model. Since axioms are nothing more than basic sets of formulae that ought to be true, we can also test axioms against such a model in principle. Before going into the examples, let us briefly describe the principles of testing axioms. More technical details will be given in §4.

Given an interpretation for function symbols and constants (i.e., model) of the logic, we can evaluate truth values of the formulae without quantifiers. For example, `plus(zero,zero)=zero` is true and `plus(one,zero)=zero` is false based on the interpretation of `one` as integer 1, `zero` as integer 0, and `plus` as the integer addition function.

We can interpret formulae with quantified variables as functions from the values of the quantified variables to truth values. For example, we can interpret `![X,Y]: plus(X,Y)=plus(Y,X)` as a function $\lambda(x, y).\, x+y = y+x$ which takes two integer pairs as input and tests whether $x+y$ is equal to $y+x$. This function will evaluate to true for any given test input $(x, y)$. When there exist test inputs under which the interpretation evaluates to false, then the original formula is invalid. For example, `![X,Y]: plus(X,Y)=X` is invalid since its interpretation $\lambda(x, y).\, x + y = x$ evaluates to false when applied to the test input $(1, 1)$.

Formulae with implication need additional care when choosing input values for testing. To avoid vacuous satisfactions of the formula we must chose inputs that satisfy the premise. In general, finding inputs satisfying the premise of a given formula requires solving equations, and for this we use a combination of the SMT solver Yices [6] and custom data generators (so-called "smart generators").

In the following subsections, we will give a high-level view of how we test the axioms with the example axioms from §2 and also some from AutoCert.

### 3.1 Testing Axioms for Numerical Arithmetic

Numeric values are one of the basic types in programming languages like C. Although the axioms on numerical arithmetic tend to be simple and small compared to other axioms (e.g., axioms on array operations) used in AutoCert, we were still able to identify some unexpected problems by testing. Those problems were commonly due to the untyped first-order logic terms being unintentionally interpreted as overloaded types. Even though the author of the axiom intended to write an axiom on one specific numeric type, say integers, that axiom could possibly apply to another numeric type, say reals.

For example, the following axiom formalizes the idea that the index of an array representing an 3-by-3 matrix uniquely determines the row and the column:

```
fof(uniq_rep_3by3, axiom,
 ! [X1, Y1, X2, Y2]: (
      ( plus(X1,times(3,Y1)) = plus(times(3,Y2),X2)
      & leq(0,X1) & leq(X1,2) & leq(0,Y1) & leq(Y1,2)
      & leq(0,X2) & leq(X2,2) & leq(0,Y2) & leq(Y2,2) )
   => (X1=X2 & Y1=Y2) ) ).
```

To test the axiom it is first translated into the following function (where we limit ourselves to primitives in the Haskell prelude library):

$$\begin{aligned}\lambda(x_1, y_1, x_2, y_2)\;.\;\neg(x_1 + 3y_1 = 3y_2 + x_2 \wedge\ & 0 \le x_1 \le 2 \ \wedge\ 0 \le y_1 \le 2 \\ \wedge\ & 0 \le x_2 \le 2 \ \wedge\ 0 \le y_2 \le 2) \\ \vee\,(x_1 = x_2 \ \wedge\ y_1 = y_2)&\end{aligned}$$

Assuming that this function is defined over integers (i.e., $x_1$, $y_1$, $x_2$, $y_2$ have integer type), we can generate test inputs of integer quadruples that satisfy the constraint of the premise $(x_1 + 3y_1 = 3y_2 + x_2 \wedge 0 \le x_1 \le 2 \wedge 0 \le y_1 \le 2 \wedge 0 \le x_2 \le 2 \wedge 0 \le y_2 \le 2)$. Since the constraint is linear, Yices can generate such test inputs automatically, and all tests succeed.

However, nothing in the axiom says that the indices must be interpreted as integers, and the axiom can just as well be interpreted using floating points, and with `plus` and `times` interpreted as the overloaded operators $+$ and $*$ in C. If we test with this interpretation we find counterexamples such as $(x_1, y_1, x_2, y_2) = (\frac{1}{2}, \frac{1}{2}, 2, 0)$. The existence of such an unintended interpretation can lead to unsoundness.

## 3.2 Testing Axioms for Arrays

Array bounds errors can cause problems in axioms as well as in programming. For example, recall the axiom `update_last` introduced in §2.

```
fof(update_last, axiom,
 ![A,I,X,Y] : update(update(A,I,X),I,Y) = update(A,I,Y) ).
```

When we give the natural interpretation to `update`, the test routine will abort after a few rounds of test inputs because the index variable `I` will go out of range.

Rather than complicate the model by interpreting the result of `update` to include a special value for out-of-bounds errors, we modify the axiom to constrain the range of the array index variable:

```
fof(update_last_in_range, axiom,
 ![A,I,X,Y]:( (leq(lo(A),I) & leq(I,hi(A)))
           => update(update(A,I,X),I,Y) = update(A,I,Y) ) ).
```

Now, all tests on `update_last_in_range` succeed since we only generate test inputs satisfying the premise `(leq(lo(A),I) & leq(I,hi(A)))`.

Similarly, we can also modify the axiom `update_commute` as follows.

```
fof(symm_joseph, axiom,
 ! [I0, J0, I, J, A, B, C, D, E, F, N, M] : (
    ( leq(0,I0) & leq(I0,N) & leq(0,J0) & leq(J0,N)
    & leq(0, I) & leq(I, M) & leq(0, J) & leq(J, M)
    & select2D(D, I, J) = select2D(D, J, I)
    & select2D(A,I0,J0) = select2D(A,J0,I0)
    & select2D(F,I0,J0) = select2D(F,J0,I0) )
 =>
    select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                     mmul(E,mmul(F,trans(E)))),
                                trans(B)))),                  I0, J0)
  = select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                     mmul(E,mmul(F,trans(E)))),
                                trans(B)))),                  J0, I0) ) ).

fof(symm_joseph_fix, axiom,
 ! [A, B, C, D, E, F, N, M] : (
   ( ( ! [I, J] : ( (leq(0,I) & leq(I,M) & leq(0,J) & leq(J,M))
                => select2D(D,I,J) = select2D(D,J,I) ) )
   & ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
                => select2D(A,I,J) = select2D(A,J,I) ) )
   & ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
                => select2D(F,I,J) = select2D(F,J,I) ) ) )
   =>
     ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
       => select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                           mmul(E,mmul(F,trans(E))) ),
                                      trans(B)))),                  I, J)
        = select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                           mmul(E,mmul(F,trans(E))) ),
                                      trans(B)))),                  J, I)
    ) ) ) ).
```

**Fig. 2.** An erroneous axiom on symmetric matrices and the fixed version

```
fof(update_commute_in_range, axiom,
![A,I,J,X,Y]:( (leq(lo(A),I) & leq(I,hi(A)) & leq(lo(A),J) & leq(J,hi(A)))
            => update(update(A,I,X),J,Y) = update(update(A,J,Y),I,X) ) ).
```

Then, we can run the tests on the above axiom without array bounds error, and in fact discover counterexamples where `I` and `J` are the same but `X` and `Y` are different. We can correct this axiom to be valid as follows by adding the additional constraint that `I` and `J` are different (i.e., either `I` is less than `J` or vice versa).

```
fof(update_commute_in_range_fixed, axiom,
 ![A,I,J,X,Y]:( (leq(lo(A),I) & leq(I,hi(A)) & leq(lo(A),J) & leq(J,hi(A))
               & (lt(I,J) | lt(J,I)) )
            => update(update(A,I,X),J,Y) = update(update(A,J,Y),I,X) ) ).
```

The test for this new axiom succeeds for all test inputs.

As for the axiom `rotation2D_def`, itself, we observed above that it is quite different from the "natural" definition of the matrix. Thus, we test the axiom against the interpretation `rot2D` in Figure 4 with 100 randomly generated arrays of size 4 and find that it does indeed pass all tests.

Finally, the axiom `symm_joseph` in Figure 2 is intended to state that $\mathbf{A} + \mathbf{B}(\mathbf{CDC}^{\mathrm{T}} + \mathbf{EFE}^{\mathrm{T}})\mathbf{B}^{\mathrm{T}}$ is a symmetric matrix when $\mathbf{A}$ and $\mathbf{F}$ are $N{\times}N$ symmetric matrices and $\mathbf{D}$ is an $M \times M$ symmetric matrix. This matrix expression, which is required to be symmetric, arises in the implementation of the Joseph update in Kalman filters. However, when we test this axiom for $N = M = 3$ and assuming `B`, `C`, and `E` are all $3 \times 3$ matrices, we get counterexamples such as

$$(\mathtt{I0}, \mathtt{J0}, \mathtt{I}, \mathtt{J}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{N}, \mathtt{M}) = \left(1, 0, 0, 0, \begin{pmatrix} 9.39 & 4.0 & -3.53 \\ 4.0 & 0.640 & -0.988 \\ -2.29 & -23.8 & -1.467 \end{pmatrix}, \ldots \right).$$

We can immediately see that something is wrong since `A` is not symmetric. The problem is that the scope of the quantifiers is incorrect and therefore does not correctly specify that the matrices are symmetric. This is fixed in `symm_joseph_fix` using another level of variable bindings for `I` and `J`, and the test succeeds for all test inputs under the same assumption that $N = M = 3$ and `B`, `C`, and `E` are all $3 \times 3$ matrices. However, `symm_joseph_fix` still shares the same index range problem as `update_last` and `update_commute`. Moreover, nothing in the axiom prevents $N$ and $M$ being negative, and the dimensions for matrices `B`, `C`, and `E` are not explicitly constrained to make the matrix operations `mmul` and `madd` well defined.

## 4 Implementation

We have implemented a tool using Template Haskell [7], QuickCheck [8], and Yices [6], as illustrated in Figure 3. An axiom in TPTP syntax is parsed and automatically translated into a lambda term using Template Haskell. Using a metaprogramming language like Template Haskell has the advantage that we inherit the underlying type system for the interpreted terms. We generate a Haskell function rather than implement an evaluator for the logic. During the translation we transform the logical formula into a "PNF (prenex normal form) like" form moving universally quantified variables to the top level as much as possible.[3] For example, `![X]:(X=0 => (![Y]:(Y=X => Y=0)))` is translated into the logically equivalent `![X,Y]:(X=0 => (Y=X=>Y=0))`. We need to lift all the variables to the top level universal quantification to interpret the axioms as executable functions.

Given a user-provided interpretation for the constants, the lambda term becomes an executable function which can then be used as a property in QuickCheck,

[3] The difference from PNF is that we avoid introducing existential quantification where possible.

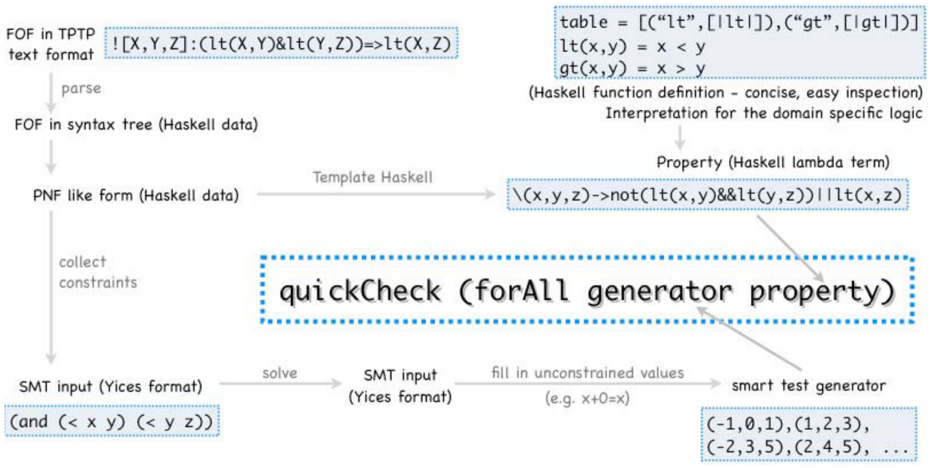

FOF in TPTP text format
![X,Y,Z]:(lt(X,Y)&lt(Y,Z))=>lt(X,Z)
parse
FOF in syntax tree (Haskell data)
PNF like form (Haskell data)
Template Haskell
collect constraints
SMT input (Yices format)
(and (< x y) (< y z))
solve
SMT input (Yices format)
fill in unconstrained values (e.g. x+0=x)
smart test generator
(-1,0,1),(1,2,3), (-2,3,5),(2,4,5), ...
table = [("lt",[|lt|]),("gt",[|gt|])]
lt(x,y) = x < y
gt(x,y) = x > y
(Haskell function definition - concise, easy inspection)
Interpretation for the domain specific logic
Property (Haskell lambda term)
\(x,y,z)->not(lt(x,y)&&lt(y,z))||lt(x,z)
quickCheck (forAll generator property)

**Fig. 3.** Testing Framework

```
pred2hsInterpTable = [ ("rot2D",[|rot2D|]), ("lt",[|lt|]), ("leq",[|leq|]) ]
term2hsInterpTable =
                    [ ("lo",[|lo|]), ("hi",[|hi|]), ("update",[|update|])
                    , ("uminus",[|uminus|]), ("cos",[|cos|]), ("sin",[|sin|])
                    , ("0",[|0|]), ("1",[|1|]), ("2",[|2|]), ("3",[|3|]) ]

rot2D :: (Array Integer Double, Double) -> Bool
rot2D(a,t) = (a!0) ===    cos t   && (a!1) === sin t
          && (a!2) === (- sin t) && (a!3) === cos t

lo a = fst( bounds a )
hi a = snd( bounds a )

uminus :: Double -> Double
uminus x = -x

update :: (Array Integer Double, Integer, Double) -> Array Integer Double
update(arr,i,c) = arr // [(i,c)]

leq(x,y) = x <= y
let(x,y) = x < y
```

**Fig. 4.** Interpretation for the 2D rotation matrix axiomatization

a property-based testing framework for Haskell. The user-provided interpretation should be concise and easy to inspect so that it can serve as a reference model, suitable for inspection by domain experts. We believe that a declarative language like Haskell is suitable because of its conciseness. For example, part of the interpretation for testing the two-dimensional rotation matrix axioms discussed in §2 and §3 is shown in Figure 4. The tables `pred2hsInterpTable` and `term2hsInterpTable` interpret each predicate and term symbol as Haskell values. Here, `("cos",[|cos|])` relates the parsed TPTP symbol `"cos"` with the Haskell library function `cos` wrapped with a Template Haskell bracket. This allows a piece of syntax to be passed around as a meta-programming object and executed later (i.e., when we invoke QuickCheck) without conflicting with the Haskell type system.

The next step is to use a combination of QuickCheck library random generators and Yices to automatically synthesize test generators that generate inputs satisfying the premises of the axiom formula, thus avoiding vacuous tests. In the case where Yices cannot solve the constraints, we use our own smart generators with the help of combinator libraries in QuickCheck.

We sometimes need to patch or fill in unconstrained values that are missing from the results of Yices. For example, among the four variables in the axiom `update_last_in_range`, `X` and `Y` are unconstrained since they do not appear in the premise, so we randomly generate `X` and `Y` independently from Yices. Sometimes, there can be unconstrained variables even if they do appear in the premise because the constraints on those variables are trivial (e.g., solutions for $x$ satisfying $x + 0 = x$).

Then, we can invoke QuickCheck over the property combined with the test generator. The basic idea is to call `quickCheck (forAll generator property)`, where `generator` generates the test data and `property` is the property to test. However, we make a few changes to this basic scheme, which we now discuss while showing how we invoke QuickCheck on some of the axioms in the Haskell interactive environment.

For the axioms on integers with linear constraints such as `uniq_rep_3by3` in §3.1, it is possible to fully automate the test. Recall that we only collect constraints from the premise (i.e., left-hand side of the implication). For example, we can run the test on `uniq_req_3by3` as follows.

```
> mQuickCheck( $(interpQints uniqr3by3) (genCtrs uniqr3by3) )

(0 tests)
non-trivial case
...
(99 tests)
non-trivial case
+++ OK, passed 100 tests.
```

The first change to the basic scheme is that `mQuickCheck` has a wrapper over the library function `quickCheck` which allows its argument to be of an IO monad type. This allows test generators to perform the side effect of communicating with the Yices process in order to solve the constraints. `interpQints` generates an interpretation for the axiom from its argument `uniqr3by3`, and `uniqr3by3` is the syntax tree for the axiom `uniq_req_3by3`. The `genCtrs` function generates constraints from the premises of the axiom.

Second, for the axioms on arrays and on types other than integers, we may need type annotations and other constraints to narrow the search space. For example, to test `update_commute_in_range` we call:

```
> mQuickCheck(
    $(interpQ updatecommr
       [| (listArray(0,3)[1.0..]::Array Integer Double,
           0::Integer, 0::Integer, 0.0::Double, 0.0::Double) |] )
    (ASSERT(Y.VarE "_I":>=LitI 0):ASSERT(Y.VarE "_J":<=LitI 3):
     ASSERT(Y.VarE "_J":>=LitI 0):ASSERT(Y.VarE "_I":<=LitI 3):
     (genCtrs updatecommr))


...
*** Failed! Falsifiable (after 4 tests):
(array (0,3) [...], 0, 0, 2.8288383471313097, 1.9408590255175935)
```

After a few tests it finds a counterexample such that both the index variables `I` and `J` are 0. The additional annotation is because of the dependencies between variables. The variables `I` and `J` in the axiom `update_commute_in_range` are constrained by the index range of `A`. This means that we can only generate useful test values of `I` and `J` after generating a test value for `A`. The type information including the array index range is specified in `[|...|]` and the `ASSERT`'s specify the constraints for the variables `I` and `J`. Currently we do not automatically infer these dependencies, but this could be done.

Lastly, when we test axioms involving floating point numbers, such as `symm_joseph`, we need to give some tolerance for errors. Otherwise, tests will fail for most of the mathematical properties (e.g., associativity of addition) we expect to hold on real number arithmetic. We define an overloaded comparison operator `(===)` in Haskell which compares integers with the usual equality operator `(==)` but compares floating point numbers with a predefined error tolerance.

## 4.1 Evaluation

In addition to testing the axioms from AutoCert's domain theory, we have also carried out some simple mutation testing on several of the axioms in order to simulate the most common errors. For example, we replace logical operators (e.g., conjunction with disjunction), change the polarity of premises (adding and removing negation), replace numeric indices (to give off-by-one errors), and switch variables and function symbols (e.g., `I` with `J`, `sin` with `cos`).

There are three possibilities: the premises are satisfiable and the mutated axiom is still valid, the premises become unsatisfiable and the axiom is vacuously true, or the mutant is invalid. We created a range of mutants for the axioms considered in this paper and, after filtering out the provable mutants we were, in each case, able to either derive a counterexample within 10 steps, or conclude that the mutated premises were vacuously true.

## 5 Related Work

The idea of evaluating propositions with respect to a computational interpretation goes back to early work of Green [9] and Weyhrauch [10]. More recently, there has been some work on the use of testing to validate and debug logical conjectures. Claessen and Svensson [11] use QuickCheck to test FOL conjectures arising in inductive proofs of protocol correctness. Propositions are interpreted as invariants on a particular state transition system. Generating test cases for invariants amounts to generating paths from a random initial state. To test inductive invariants they "adapt" an arbitrarily chosen (possibly non-reachable) state to the proposition-under-test, effectively giving a test data generator generator. Berghofer and Nipkow [12] also use QuickCheck, to test theorems in Isabelle/HOL, particularly those involving inductive data types and inductive predicates. They create generators to generate data of arbitrary size for any inductive data type. In both these cases, the authors' goals are to test conjectures in a logic, rather than the axioms of the underlying logic itself, given a computational model.

Carlier and Dubois [13] have similar motivation and approach to ours, but in the setting of a typed functional language and a higher-order proof assistant. Since they mostly rely on random testing they generate and discard many test cases before they collect meaningful test cases. In contrast, we try to generate tests data efficiently by automatically synthesizing smart generators. Dybjer et al. [14] explore testing and proving in a dependent type setting, but do not automate the synthesis of test generators.

Planware [15] is a system for the deductive synthesis of planning and scheduling software. In deductive synthesis, implementations are synthesized from specifications through a sequence of correctness-preserving refinements. Correctness of these steps ultimately rests on a logical axiomatization of the domain theory. In Planware, the axioms are validated [16] via a theory morphism, that is, by translation into conjectures in another logic, in this case, set theory, where they are proven as theorems. The target theory thus serves as the intended interpretation.

Theory development in Isabelle [17] also typically proceeds in such a "definitional" style, where more complex properties are built from a small set-theoretic core. However, we have not adopted this approach since we consider the domain-specific axioms (in contrast to the underlying laws of arithmetic and relational algebra) to be our starting point. These definitions and laws, typically coming from mission documents, are thus tantamount to requirements. Moreover, it would be a lot of work to derive them from first principles, and would provide little benefit to engineers.

## 6 Conclusion

We have described our approach to model-based testing of first-order logic axioms used by the verification tool AutoCert. We believe that our approach can help to systematically debug axioms, and also help maintain soundness of the logic while actively developing axioms. We have shown that it is quite feasible to derive counterexamples, even when the axioms are difficult to inspect. The computational model serves both as an interpretation against which the axioms can be tested, and as a reference which can be inspected by domain experts, since they remain significantly clearer than the axiomatization, particularly when we optimize the axioms to make the theorem provers search for proofs more efficiently. One clear conclusion we draw is the need for a *typed* logic to reduce unsoundness. Although types can be encoded in an untyped setting, we plan to investigate the recently proposed Typed First-Order Form [18].

Previously, we had frequently run into inconsistency, and sometimes this was not noticed until quite some time after the erroneous axioms had been added. Using the testing framework we have been able to find counterexamples for some axioms that had been previously known to be suspicious, as well as some previously unsuspected axioms. It also helped us avoid unsoundness arising from implicit but different models of the logic. Testing and proving are therefore complementary aspects to developing a formal verification.

An important aspect of testing is discovering corner cases of idealized models (e.g., overflow in fixed point arithmetic and round-off errors in floating point arithmetic). In our work, we used an arbitrary tolerance for round-off errors, but a more sophisticated notion, depending on input variables, is appropriate.

In terms of developing an infrastructure for the certification of safety-critical software, minimizing the trusted base is important. An important part of testing, and thus qualifying, the axioms will be to develop an appropriate notion of *coverage* (as in [13]), to give some measure of confidence that enough testing has been done. In the case of testing programs, coverage criteria are usually expressed in terms of branches and decisions taken by the software. For axioms, we also aim to cover all branches (that is, all independent ways of satisfying the hypotheses) as well as covering the domain (e.g., by considering all representatives of each frame of a DCM).

We are currently extending the framework in two directions: testing verification conditions, and testing functions. As observed by Claessen and Svensson [11], we would like to know when a VC really is invalid, and when it is simply unprovable due to a missing axiom, say. We are extending the framework to be able to also test VCs, and thus provide insight into when the problem lies in a missing axiom, rather than an invalid VC. A related goal is to black-box test library functions which implement the concepts in the axioms, using the same mathematical specifications.

Lastly, we have also tested a number of axioms that involve physical units and equations. These axioms need to be modified in order to make them testable, but we believe that this can be done in a principled and systematic manner.

## References

1. Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conference (March 2008)

2. Sutcliffe, G.: System description: System On TPTP. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 406–410. Springer, Heidelberg (2000)
3. Sutcliffe, G., Denney, E., Fischer, B.: Practical proof checking for program certification. In: Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning, ESCAR 2005 (July 2005)
4. Kuipers, J.B.: Quaternions and Rotation Sequences. Princeton University Press, Princeton (1999)
5. Vallado, D.A.: Fundamentals of Astrodynamics and Applications, 2nd edn., Space Technology Library. Microcosm Press/Kluwer Academic Publishers (2001)
6. Dutertre, B., de Moura, L.: The YICES SMT solver (2006), Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`
7. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In: ACM SIGPLAN Haskell Workshop 2002, October 2002, pp. 1–16. ACM Press, New York (2002)
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, September 2000, pp. 268–279 (2000)
9. Green, C.: The Application of Theorem Proving to Question-Answering Systems. PhD thesis, Stanford University (1969)
10. Weyhrauch, R.: Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence 13(1,2), 133–170 (1980)
11. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 48–65. Springer, Heidelberg (2008)
12. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004), pp. 230–239 (2004)
13. Carlier, M., Dubois, C.: Functional testing in the Focal environment. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 84–98. Springer, Heidelberg (2008)
14. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer, Heidelberg (2003)
15. Blaine, L., Gilham, L., Liu, J., Smith, D., Westfold, S.: Planware: Domain-specific synthesis of high-performance schedulers. In: The 13th IEEE International Conference on Automated Software Engineering (ASE 1998), Honolulu, Hawaii, USA, pp. 270–280. IEEE Computer Society, Los Alamitos (1998)
16. Becker, M., Smith, D.R.: Model validation in Planware. In: Verification and Validation of Model-Based Planning and Scheduling Systems (VVPS 2005), Monterey, California, USA, June 6-7 (2005)
17. Paulson, L., Nipkow, T.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
18. Claessen, K., Sutcliffe, G.: A simple type system for FOF (2009), `http://www.cs.miami.edu/~tptp/TPTP/Proposals/TypedFOF.html`

# Proving and Visualizing OCL Invariant Independence by Automatically Generated Test Cases

Martin Gogolla, Lars Hamann, and Mirco Kuhlmann

Universität Bremen, Informatik, AG Datenbanksysteme, D-28334 Bremen

**Abstract.** Within model-driven development, class invariants play a central role. An essential property of a collection of invariants is the independence of each single invariant, i.e., the invariant at hand cannot be deduced from the other invariants. The paper explains with three example models the details of an approach for automatically proving and representing invariant independence on the basis of a script constructing large test cases for the underlying model. Analysis of invariant independence is visualized by means of several diagrams like a 'test configuration and result' diagram, an 'invariant dependence detail' diagram, and an 'invariant dependence overview' diagram. The paper also discusses how to build the test case construction script in a systematic way. The test case construction script is written by the model developer, but a general construction frame for the script is outlined.

## 1 Introduction

Software development with model-driven techniques and languages like the Unified Modeling Language (UML) and the Object Constraint Language (OCL) has attracted many practitioners and researchers in recent years. In object-oriented approaches, class invariants play a central role. A crucial property of a collection of invariants is the independence of each single invariant, i.e., a considered invariant cannot be deduced from the other stated invariants. This property may be viewed as guaranteeing the absence of redundancy in an invariant set.

This paper explains an approach for automatically proving and representing invariant independence on the basis of a script constructing large test cases for the underlying model. When we use the notion script here, we always refer to such a test case construction script which belongs to the modeling level and not to a detail of the implementation. The test case construction script is written by the model developer, but a general construction frame for the script is outlined here in order to formulate a primary version of the script. Development is supported and visualized by means of several diagrams allowing to represent test configurations and results. The proposed diagrams can be generated in an automatic way from the executed test cases. This paper also explains how to write a test case construction script in general. Our work is done within the

UML-based Specification Environment (USE), a modeling and validation system [GBR05, GHK07]. The USE details of all test cases in this paper may be found in [GHK10]. One motivation for introducing the various diagram forms here was to abstract away from test case details.

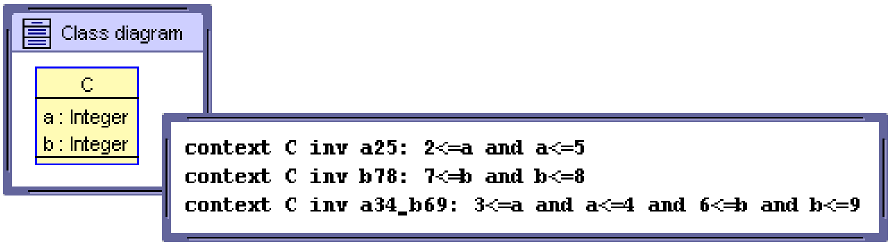
Invariant independence in models is an important property in object-oriented approaches, because if it holds, then all invariants have to be respected directly in an implementation; if it does not hold, then it is desirable to know, why it does not hold and which invariants contribute to redundancy. Our proposal helps in answering such questions and can be used to detect which subsets of the given invariants are independent. Invariant redundancy often occurs when important system properties are consequences of other technical requirements. Therefore techniques to detect dependencies are valuable.

The rest of the paper is structured as follows. Section 2 introduces the necessity for invariant negation, affirmation, and deactivation in test case construction scripts and shows the different diagram forms. Section 3 explains with a more complex example test case construction and shows more involved invariant dependencies. Section 4 discusses an example from the literature and puts forward the general template for a test case construction script. The paper ends with a discussion of related work and a conclusion. The generated test cases in [GHK10] may be seen as appendix-like add-on for the paper.

## 2 Basics of Invariant Independence

As an extension to the research presented in [GHK09] we here propose a more sophisticated way of producing test cases which prove the independence of invariants. This paper employs invariant deactivation, a technique which we did not apply before for showing invariant independence. By doing so, we are able to obtain more independence relationships. Furthermore we represent the different test case configurations and test results in diagrammatic form by means of so-called 'test configuration and result', 'invariant dependence detail', and 'invariant dependence overview' diagrams.

The artificial example model shown in Fig. 1 possessing a single class C and two integer attributes a and b demonstrates the new features. The three invariants restrict the attribute values and partly overlap.



```
context C inv a25: 2<=a and a<=5
context C inv b78: 7<=b and b<=8
context C inv a34_b69: 3<=a and a<=4 and 6<=b and b<=9
```

**Fig. 1.** Example Model CAB

The following test case construction script enumerates possibilities for attribute values until a system state satisfying the current invariant configuration is found. A system state is also called snapshot because it represents the state of the system at a particular point during the dynamic development of the system. With the UML, object diagrams are employed for representing snapshots. An invariant can be negated, affirmed, or deactivated. The script language ASSL (A Snapshot Sequence Language [GBR05]) is a hybrid language which uses elements of imperative programming and OCL. OCL and ASSL are decidable because quantification is applied only on finite domains. In this example, the bounds for the attribute values can even be derived systematically from the constraints which use the constant 2 as the minimum integer and the constant 9 as the maximum integer. The bounds in the script extend the used bounds in the constraints by 1 on both interval ends in order to cover positive and negative test cases.

```
procedure genWorld()
var c: C;
begin
c:=[C.allInstances()->any(true)];
[c].a:=Try([Sequence{1..10}]);
[c].b:=Try([Sequence{1..10}]);
end;
```

The first step for the analysis of invariant independence is to call the test case generation script as often as determined by the number of invariants: If there exist n invariants, then the script is called n times; each call sets exactly one invariant as negated and all other invariants as affirmed:[1]

```
gen flags [a25 +n] [b78 -n] [a34_b69 -n]; gen start genWorld()
gen flags [a25 -n] [b78 +n] [a34_b69 -n]; gen start genWorld()
gen flags [a25 -n] [b78 -n] [a34_b69 +n]; gen start genWorld()
```

The different calls are represented in the Test Configuration and Test Result diagram in Fig. 2. Each call is shown as a line of invariants together with an indication expressing whether a snapshot has been found or not. Affirmed, negated and deactivated invariants are displayed differently (in the calls in Fig. 2 there are no deactivations, but such deactivations will show up below). Furthermore, if a snapshot has been found, an arrow indicating independence resp. non-implication may be drawn from each affirmed invariant to a negated invariant. For example, the upper-left non-implication arrow in Fig. 2 expresses that we have formally shown the fact (theorem): `not(a25 implies b78)`.

The currently known relationships between invariants are displayed in the Invariant Dependence Detail diagram in Fig. 3. All known non-implications are shown with non-implication arrows, and unknown relationships are shown as grey arrows.

[1] In our current implementation, the syntax for invariant affirmation, negation and deactivation is slightly different; the flags for invariant affirmation, negation and deactivation must be stated for each each single invariant (see [GHK10]).

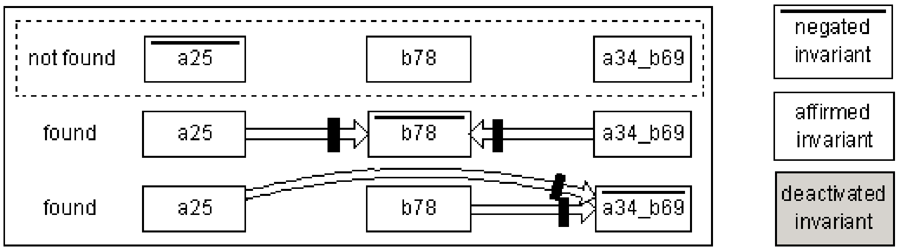


**Fig. 2.** Test Configuration and Test Result Diagram (CAB-A)

In the second step for the analysis of invariant independence we study those test cases in more detail where the first step did not find snapshots. If a call with a negated invariant and with all other invariants affirmed was not successful, the configuration is extended so that all those test case generation calls are performed where at least one of the other invariants is deactivated and at least one is negated:

```
gen flags [a25 +n] [b78 +d] [a34_b69 -n]; gen start genWorld()
gen flags [a25 +n] [b78 -n] [a34_b69 +d]; gen start genWorld()
```

In the example this leads to the Test Configuration and Test Result diagram and the Invariant Dependence Detail diagram show in Fig. 4. The grey arrow in

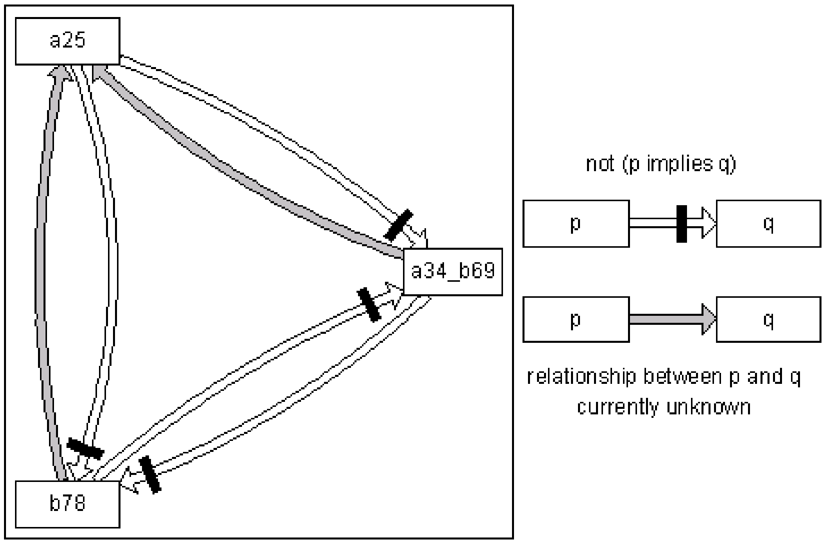


**Fig. 3.** Invariant Dependence Detail Diagram (CAB-B)

Fig. 3 from b78 to a25 has been updated to a non-implication arrow. This could only be achieved by invariant deactivation which was not considered in [GHK09].

It is worth to discuss why deactivation finds more counter examples: We assume one fixed ASSL script is employed; this script enumerates a particular finite set of system states, called the search space; first, the script is called as many times as invariants exist with exactly one invariant being negated; each single call may produce as result either a counter example or the answer that no system state exists in the search space under the current configuration for affirmed and negated invariants; in the last case the complete finite search space has been considered; only in this situation deactivation is used as a second step; within the search space, there may be counter examples which satisfy the current configuration of affirmed and negated invariants except the deactivated invariant which is violated; in order to find these counter examples, deactivation is beneficial.

The current knowledge about the invariant dependence is now aggregated into a single Invariant Dependence Overview diagram as shown in Fig. 5. All invariants are displayed with solid or dashed rectangles within a rounded rectangle. If an invariant is independent from all other invariants, it is also displayed outside the rounded rectangle with a non-implication arrow from the rounded rectangle

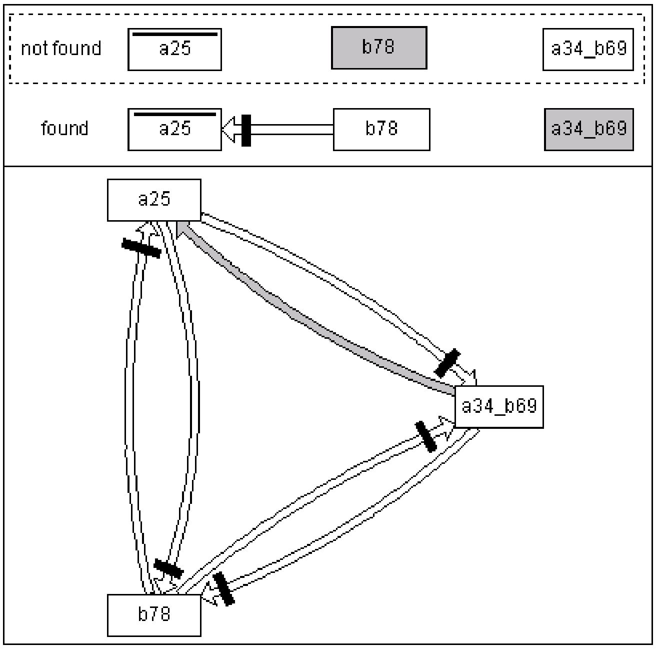


**Fig. 4.** Test Configuration and Test Result Diagram (CAB-C) as well as Updated Invariant Dependence Detail Diagram (CAB-D)

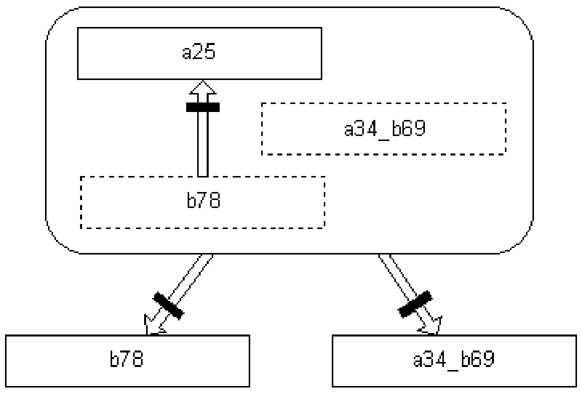


**Fig. 5.** Invariant Dependence Overview Diagram (CAB-E)

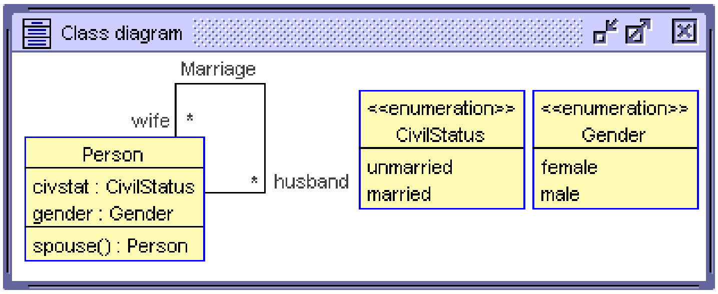
to it (for example, `b78`). This arrow means that this invariant is independent from the complete set of invariants except the invariant itself: The independent invariant is shown with a dashed rectangle within the rounded rectangle (again, for example, `b78`) in order to distinguish it from an invariant which is not independent from the other invariants (for example, `a25`); naturally, we have, for example, `b78 implies b78` and we do not have `not(b78 implies b78)`. Further non-implications may be shown within the rounded rectangle (for example, the non-implication arrow from `b78` to `a25`). This kind of abstracting details of edges is similar to the abstraction made in statecharts [Har87] for displaying different transitions as a single transition.

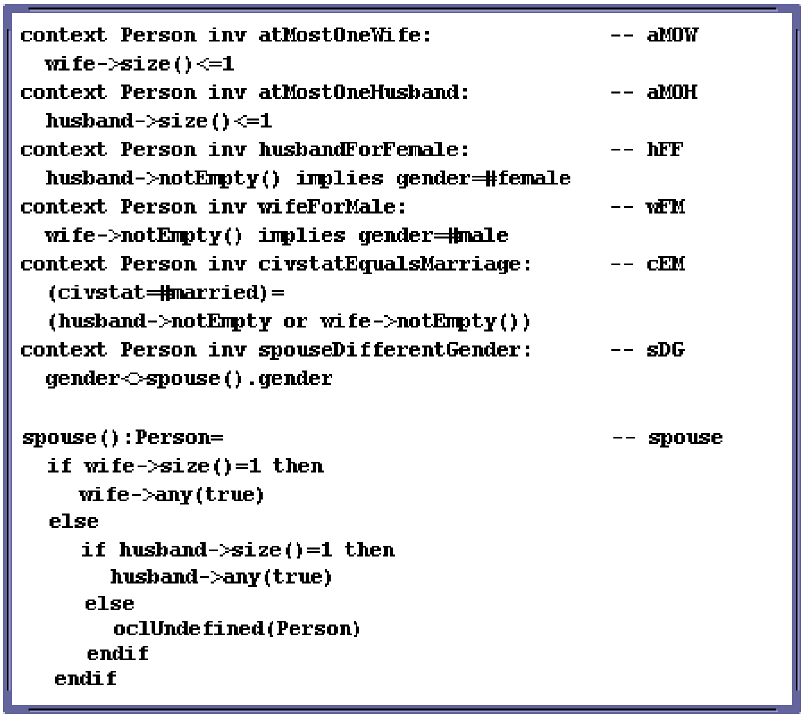
## 3 Invariant Independence in More Complex Models

Up to now we did not handle associations in test case generation scripts. The example in this section treats associations, and it shows that the fact that non-implications cannot be established is a good indicator that mutual dependencies between invariants do exist.

The example model in Fig. 6 handles the civil status of persons and constrains the attributes and links. The first two constraints are requirements which could also be expressed as multiplicity restrictions in the class diagram. However, we have formulated them explicitly in OCL because we want to check these constraints with respect to independence to the other constraints. The following test case generation script restricts the number of persons to three.
The first Test Configuration and Test Result diagram in Fig. 7 shows independence of three invariants. In each of the six calls to the test case generation script exactly one invariant is negated and the other invariants are affirmed. The test case generation script does not find examples for the three other invariants. The achieved Invariant Dependency Overview diagram is shown in Fig. 8.

The second Test Configuration and Test Result diagram in Fig. 9 with deactivations adds six non-implications. These are pictured in the Invariant Dependence Detail diagram in Fig. 10. In contrast to the example from the previous





```
context Person inv atMostOneWife:                    -- aMOW
  wife->size()<=1
context Person inv atMostOneHusband:                 -- aMOH
  husband->size()<=1
context Person inv husbandForFemale:                 -- hFF
  husband->notEmpty() implies gender=#female
context Person inv wifeForMale:                      -- wFM
  wife->notEmpty() implies gender=#male
context Person inv civstatEqualsMarriage:            -- cEM
  (civstat=#married)=
  (husband->notEmpty or wife->notEmpty())
context Person inv spouseDifferentGender:            -- sDG
  gender<>spouse().gender

spouse():Person=                                     -- spouse
  if wife->size()=1 then
    wife->any(true)
  else
    if husband->size()=1 then
      husband->any(true)
    else
      oclUndefined(Person)
    endif
  endif
```

**Fig. 6.** Example Model CIVSTAT

section, the non-implications have a single invariant as source and two invariants as target[2]. Furthermore, the three invariants husbandForFemale, wifeForMale and spouseDifferentGender, whose independence cannot be proved, show symmetrical properties in Fig. 10. This could be seen as an indication for dependencies between these invariants. Indeed, we have `(hFF and wFM) implies sDG`. This could formally be proved with approaches like [BW09] or [BHS07].

This example also shows our general strategy for tackling invariant independence. One should first show as many as possible independencies resp. non-

[2] Now we have `not(A implies B)` as well as `not(A implies C)`, but previously we had `not(B implies A)` as well as `not(C implies A)`.

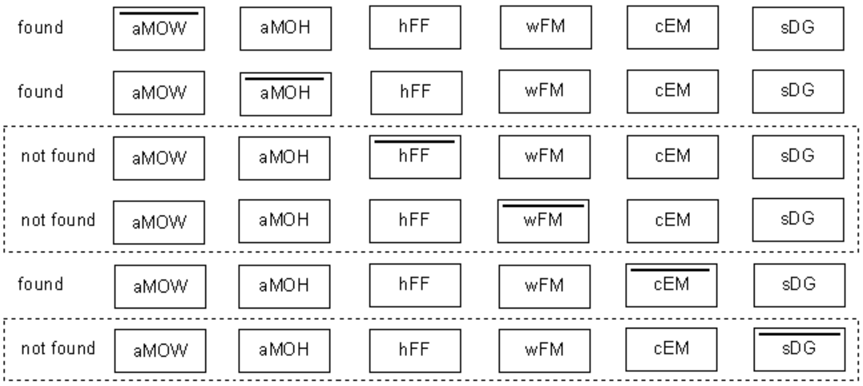


**Fig. 7.** Test Configuration and Test Result Diagram (CIVSTAT-A)

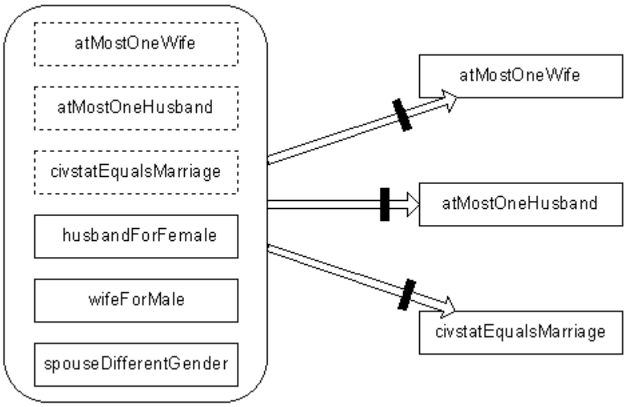


**Fig. 8.** Invariant Dependence Overview Diagram (CIVSTAT-B)

implications. Invariant dependencies must be in the complement of the shown independencies. Afterwards, the hopefully small complement can be tackled with explicit proof techniques like [BHS07, BW09].

## 4 Sanity-Check and Test Case Generation Template

The last model is an example which has not been developed by us, but which can be found in the literature [$CGQ^+08$]. The example is a sanity-check thatour proposal works for models which are developed not in our group. The specific test case generation script also shows the general structure for such a script. The script has to be written by the model developer for each individual model, but the general structure can be employed for many models and helps to formulate a primary version of the script. If this version is not efficient, it may be taken as the starting point for further improvements, for example, by using restricting OCL expressions in 'Any' or 'Try' statements which reduce the search space.

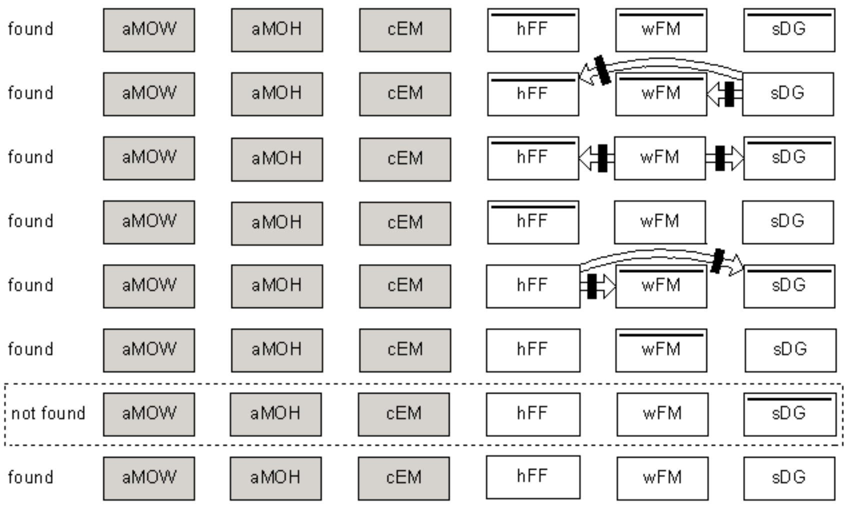


**Fig. 9.** Test Configuration and Test Result Diagram (CIVSTAT-C)

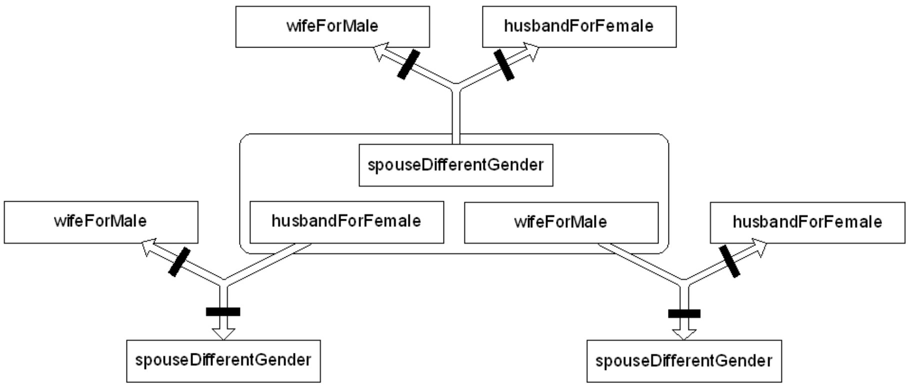
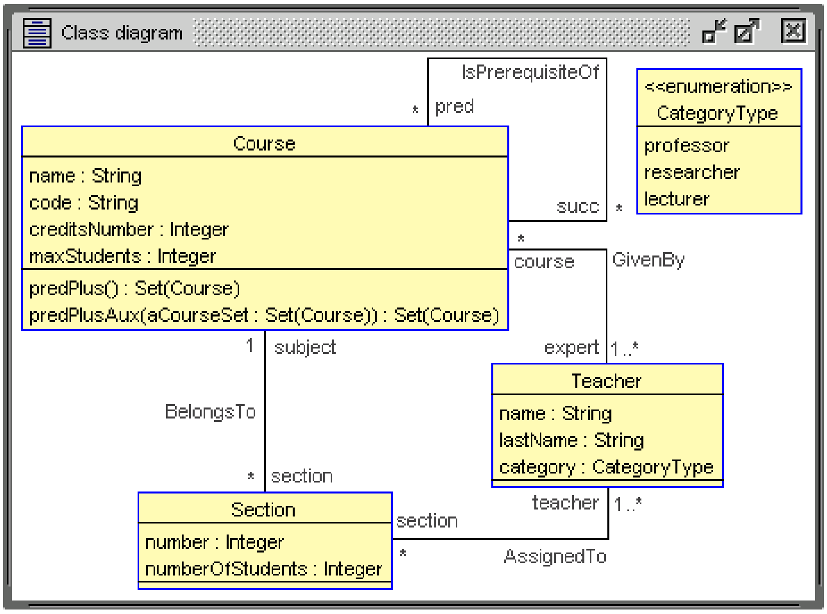


**Fig. 10.** Invariant Dependence Detail Diagram (CIVSTAT-D)



```
context Course inv nameUnique:
  Course.allInstances()->isUnique(name)
context Course inv codeUnique:
  Course.allInstances()->isUnique(code)
context Teacher inv nameLastNameUnique:
  Teacher.allInstances()->isUnique(Sequence{name,lastName})
context Section inv courseNumberUnique:
  Section.allInstances()->isUnique(Sequence{subject,number})
context Course inv isPrerequisiteOfIsAcyclic:
  self.predPlus()->excludes(self)
context Course inv limitOnCredits:
  self.creditsNumber<=10
context Course inv sectionTeacherMustBeExpert:
  self.expert->includesAll(self.section.teacher)
context Section inv limitOnNumberOfStudents:
  self.numberOfStudents<=self.subject.maxStudents
context Course inv atLeastALecturer:
  self.expert.category->includes(#lecturer)

Course::predPlus():Set(Course)=
  self.predPlusAux(self.pred)
Course::predPlusAux(aCourseSet:Set(Course)):Set(Course)=
  let oneStep=aCourseSet.pred->asSet() in
  if oneStep->exists(c|aCourseSet->excludes(c))
    then self.predPlusAux(aCourseSet->union(oneStep))
    else aCourseSet endif
```

**Fig. 11.** Example Model CST (Course-Section-Teacher)

```
procedure genWorld()
var num: Integer, numC: Integer, numT: Integer, numS: Integer,
    courses: Sequence(Course),
    teachers: Sequence(Teacher),
    sections: Sequence(Section);
begin num:=[2]; numC:=Try([Sequence{1..num}]);
numT:=Try([Sequence{1..num}]); numS:=Try([Sequence{1..num}]);
courses:=CreateN(Course,[numC]);
teachers:=CreateN(Teacher,[numT]);
sections:=CreateN(Section,[numS]);
Try(IsPrerequisiteOf,[courses],[courses]);
Try(GivenBy,[courses],[teachers]);
Try(BelongsTo,[courses],[sections]);
Try(AssignedTo,[teachers],[sections]); for i:Integer in
[Sequence{1..numC}] begin
  [courses->at(i)].name:=Try([Sequence{'DBS Course','SWE Course'}]);
  [courses->at(i)].code:=Try([Sequence{'dbs','swe'}]);
  [courses->at(i)].maxStudents:=Try([Sequence{40}]);
  [courses->at(i)].creditsNumber:=Try([Sequence{10,12}]);
  end;
for i:Integer in [Sequence{1..numT}] begin
  [teachers->at(i)].name:=Try([Sequence{'Ada','Bob'}]);
  [teachers->at(i)].lastName:=Try([Sequence{'Smith'}]);
  [teachers->at(i)].category:=Try([Sequence{#lecturer,#researcher}]);
  end;
for i:Integer in [Sequence{1..numS}] begin
  [sections->at(i)].number:=Try([Sequence{100,110}]);
  [sections->at(i)].numberOfStudents:=Try([Sequence{30,50}]);
  end;
end;
```

The model in Fig. 11 is a bit more complex with three classes, four associations and nine invariants. In the original paper, independence of invariants is discussed under the notion 'absence of redundancy' and sufficient criteria for it are stated.

According to the increased complexity of the underlying model, the test case construction script also becomes more involved. Objects, links, and attribute values are generated in that order as follows.
According to our approach, we first call the above test case generation script nine times. In each call, exactly one invariant is negated and all other are affirmed. In this case, all invariants are independent and this can be proved already with these first nine calls. The result is represented as an Invariant Dependence Overview diagram in Fig. 12. The found counter examples are shown in Fig. 13 and 14. In our view, these counter examples show the invariant independence, but they also demonstrate nicely how the models work and what the effects of the model look like in term of system states resp. object diagrams.

From the above script we can distill a general format for a test case generation script (Fig. 15). The general format defines a script which works as follows: For each class determine the maximal number of objects in that class and create

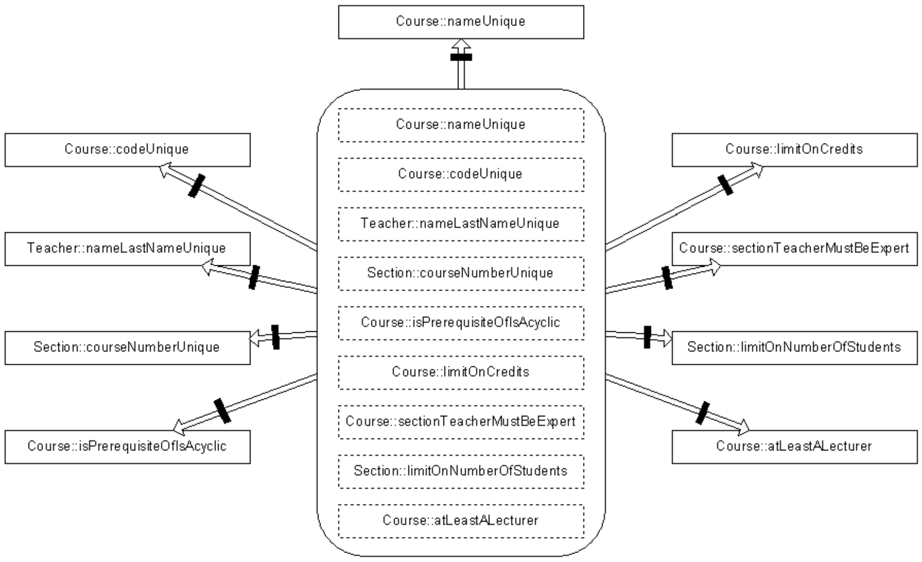

Course::nameUnique
Course::nameUnique
Course::codeUnique
Teacher::nameLastNameUnique
Section::courseNumberUnique
Course::isPrerequisiteOfIsAcyclic
Course::limitOnCredits
Course::sectionTeacherMustBeExpert
Section::limitOnNumberOfStudents
Course::atLeastALecturer
Course::codeUnique
Teacher::nameLastNameUnique
Section::courseNumberUnique
Course::isPrerequisiteOfIsAcyclic
Course::limitOnCredits
Course::sectionTeacherMustBeExpert
Section::limitOnNumberOfStudents
Course::atLeastALecturer

**Fig. 12.** Invariant Dependence Overview Diagram (CST-A)

new objects accordingly; thus the test case generation process will first try to populate the class with one object, then with two objects and so forth; for each association try all possible link sets in that association; for each attribute in each class try all possibilities from an explicitly given sequence of attribute values. In general, one should be conservative with regard to the maximal number of objects. One should start with lowest values and try to increase these numbers with experiments.

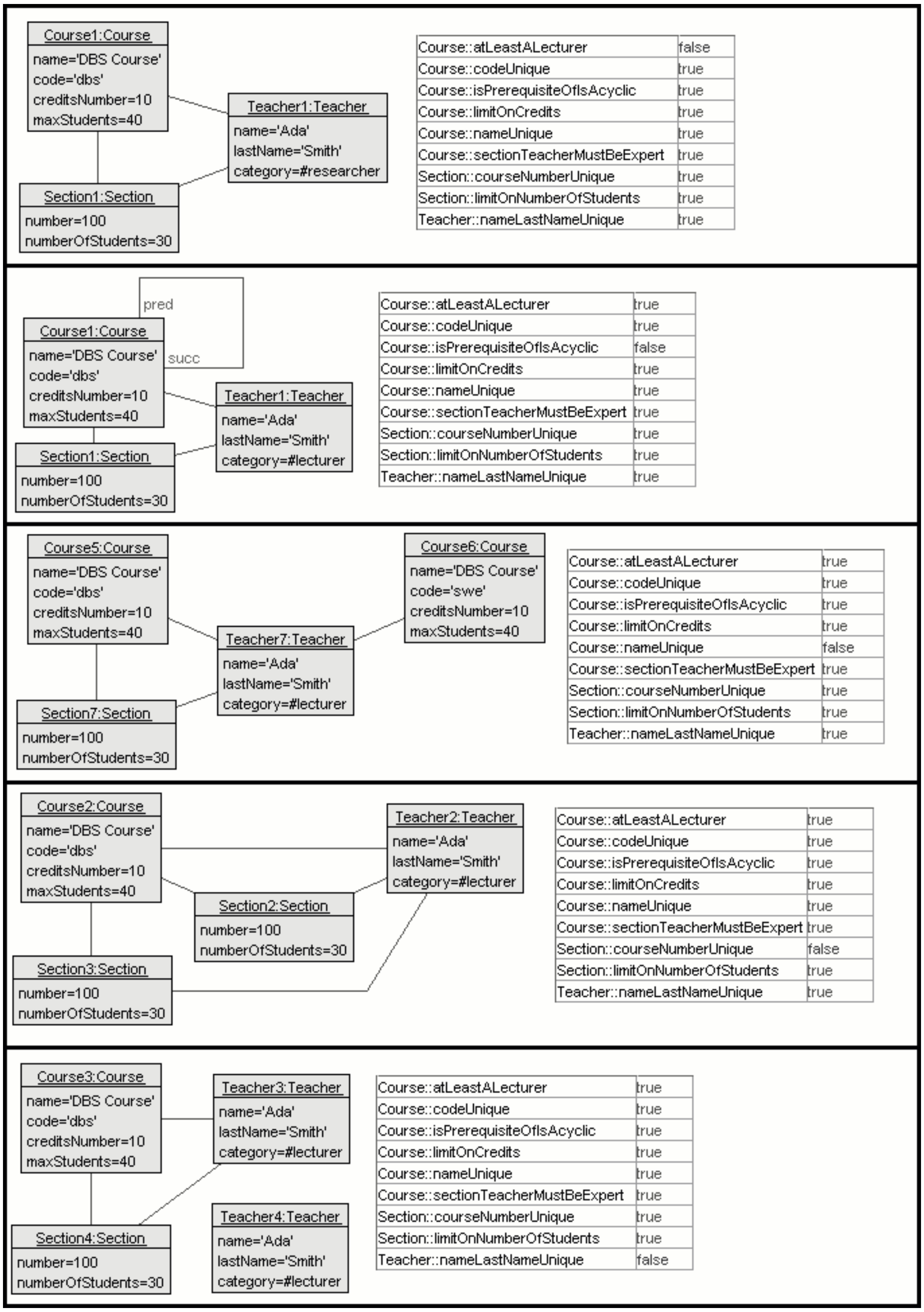
Finally, we want again point to the fact that the diagrams which we have presented in this paper are abstractions of the numerous test cases resp. snapshots which were generated in an automatic way. As is detailed in [GHK10], for the CAB model 250 snapshots, for the CIVSTAT model 101.001 snapshots, and for the CST model 2.393.734 snapshots were considered.[3]

In our examples, the test generation script produces the results within acceptable answer times on a standard laptop. However, for larger examples with more classes SAT-based techniques can produce the counter examples with faster answer times [SWK$^+$10]. As future work, we will combine the proposed visualization techniques of this paper with the more efficient SAT-like counter example production.

## 5 Related Work

Proving constraint independence has connections to other relevant approaches. Basic concepts for formal testing can be found in the pioneering paper [Gau95].

[3] This can be traced by considering the messages ‘Checked ... snapshots’ in [GHK10].



| | |
|---|---|
| Course::atLeastALecturer | false |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

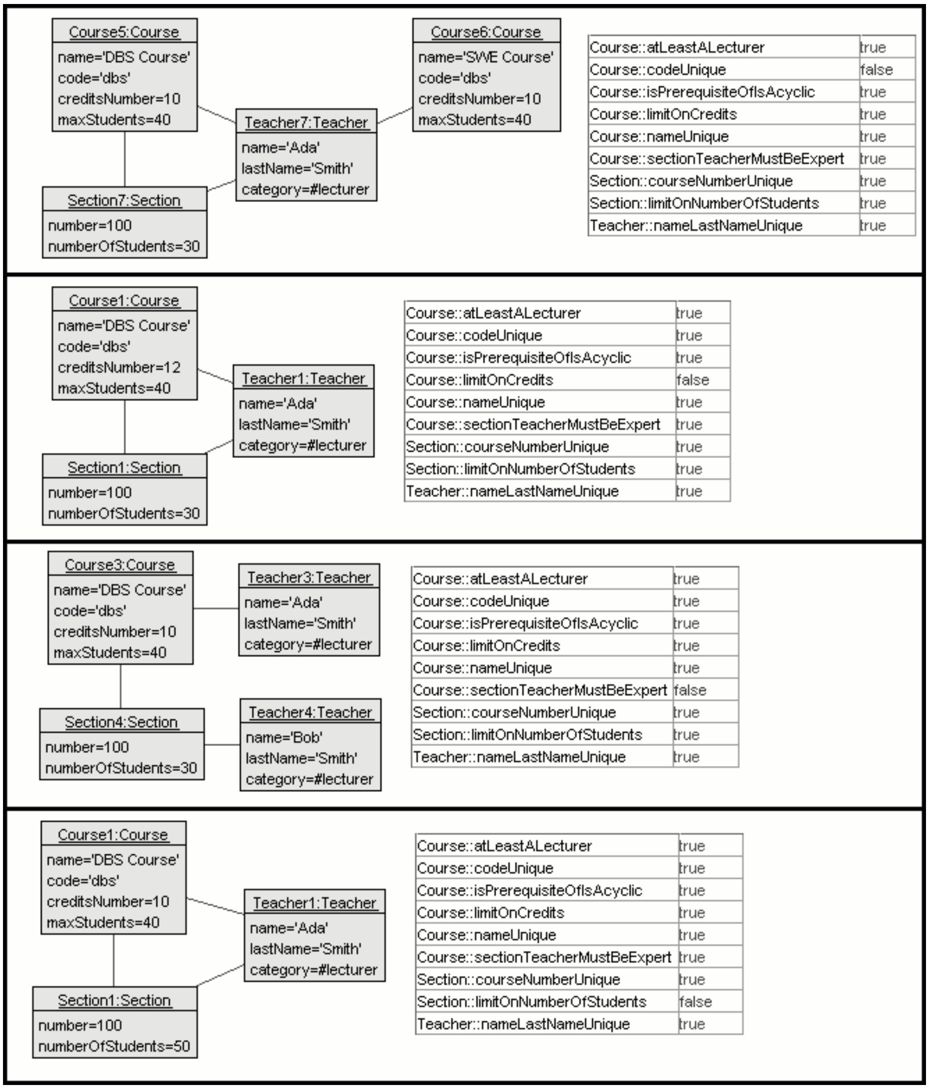
| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | false |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | false |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | false |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | false |

**Fig. 13.** Counter Examples for Model CST Proving Full Independence (Part 1)



| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | false |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | false |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | false |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | true |
| Teacher::nameLastNameUnique | true |

| | |
|---|---|
| Course::atLeastALecturer | true |
| Course::codeUnique | true |
| Course::isPrerequisiteOfIsAcyclic | true |
| Course::limitOnCredits | true |
| Course::nameUnique | true |
| Course::sectionTeacherMustBeExpert | true |
| Section::courseNumberUnique | true |
| Section::limitOnNumberOfStudents | false |
| Teacher::nameLastNameUnique | true |

**Fig. 14.** Counter Examples for Model CST Proving Full Independence (Part 2)

Independence has been studied also in connection with relational database schemata [Orm97, OS98]. The presented tool in [SCH01] allows static and dynamic model checking with focus on UML statecharts. UMLAnT [TGF+05] allows designers and testers to validate UML models in a way similar to xUnit tests. UMLAnT delegates the validation of invariants, pre- and postconditions to USE [GHK07]. Automatic test generation based on subsets of UML and OCL are examined in [BGL+07] and [AS05], whereas runtime checking of JML assertions transformed from OCL constraints is treated in [AFC08]. Approaches for static UML and OCL model verification without generating a system state

```
procedure genWorld()
var <declaration-of-variables-for-object-sequences>,
    <declaration-of-auxiliary-variables>;
begin
------------------------------------------------------------------ objects
numClass_1:=Try([Sequence{1..numClass_1Max}]);   ...
numClass_k:=Try([Sequence{1..numClass_kMax}]);
objectsClass_1:=CreateN(Class_1,[numClass_1]);   ...
objectsClass_k:=CreateN(Class_k,[numClass_k]);
-------------------------------------------------------------------- links
Try(Assoc_1,[objectsClass_i],[objectsClass_j]);   ...
Try(Assoc_m,[     ....     ],[     ....     ]);
--------------------------------------------------------- attribute values
for i:Integer in [Sequence{1..numClass_1}] begin
  [objectsClass_1->at(i)].att_1:=Try([Sequence{value_1, ..., value_p}]);
  ...
  [objectsClass_1->at(i)].att_q:=Try([            ....            ]);
  end;
<attribute-handling-for-further-classes> end;
```

**Fig. 15.** General Format for Test Case Generation Script

can be found in [MKL02], [BC06] and [BHS07]. In [MKL02], UML models and corresponding OCL expressions are translated into B specifications. The transformation result is analyzed by an external tool. The work in [BC06] focuses on static verification of dynamic model parts (sequence diagrams) with respect to the static specification. The KeY approach [BHS07] targets Java Card applications which can be verified against UML and OCL models. Our approach for checking model properties and in particular for finding models has many similarities with the Alloy [Jac06] approach. As our approach, the ALLOY approach detects model properties by setting up a search space for model snapshots and exhaustively stepping through this search space [SYC+04]. The achieved results are always relative to this search space. Thus exhaustive testing techniques on the code level [JLDM09] with methods like sparse test generation, structural test merging or oracle-based test clustering may contribute to the improvement of our approach.

## 6 Conclusion

We have proposed an approach for detecting independencies among class invariants. We were able to trace more independence facts by using invariant deactivation. The paper discussed the representation and documentation of test case generation with diagrams offering different degrees of detail. A general template for a test case generation script has been put forward.

Future work will extend our approach for handling OCL invariants to handling OCL pre- and postconditions. Special interest must be paid to the interplay between invariants and pre- and postconditions. One can improve the efficiency of our proposal with SAT-based techniques. An implementation of the proposed diagrams and case studies giving user-feedback have to be carried out. In general, we are interested to study further relationships between proving and testing in connection with UML and OCL models.

## References

[AFC08] Avila, C., Flores, G., Cheon, Y.: A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In: Arabnia, H.R., Reza, H. (eds.) Software Engineering Research and Practice, pp. 403–408. CSREA Press (2008)

[AS05] Aichernig, B.K., Salas, P.A.P.: Test Case Generation by OCL Mutation and Constraint Solving. In: QSIC, pp. 64–71. IEEE Computer Society, Los Alamitos (2005)

[BC06] Baruzzo, A., Comini, M.: Static Verification of UML Model Consistency. In: Hearnden, D., Süß, J.G., Baudry, B., Rapin, N. (eds.) Proc. 3rd Workshop on Model Design and Validation, University of Queensland, pp. 111–126 (2006)

[BGL$^+$07] Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A Subset of Precise UML for Model-Based Testing. In: A-MOST, pp. 95–104. ACM, New York (2007)

[BHS07] Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)

[BW09] Brucker, A.D., Wolff, B.: Semantics, Calculi, and Analysis for Object-Oriented Specifications. Acta Informatica 46, 255–284 (2009)

[$CGQ^+$08] Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Improving the Definition of General Constraints in UML. Journal of Software and System Modeling 7, 469–486 (2008)

[Gau95] Gaudel, M.C.: Testing can be Formal, too. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)

[GBR05] Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4, 386–398 (2005)

[GHK07] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)

[GHK10] Gogolla, M., Hamann, L., Kuhlmann, M.: Detailed Test Cases for 'Proving OCL Invariant Independence by Automatically Generated Test Cases' (2010), http://www.db.informatik.uni-bremen.de/publications/intern/GHK2010ADDON.pdf

[GHK09] Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)

[Har87] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 231–274 (1987)

[Jac06] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)

[JLDM09] Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the costs of bounded-exhaustive testing. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 171–185. Springer, Heidelberg (2009)

[MKL02] Marcano-Kamenoff, R., Lévy, N.: Transformation Rules of OCL Constraints into B Formal Expressions. In: Workshop Critical Systems Development (CSDUML 2002) with UML at 5th Int. Conf. Unified Modeling Language (2002)

[Orm97] Orman, L.V.: Relational Database Constraints as Counterexamples. Acta Informatica 34, 167–189 (1997)

[OS98] Oakasha, H., Saake, G.: Foundations for Integrity Independence in Relational Databases. In: FMLDO, pp. 143–165 (1998)

[SCH01] Shen, W., Compton, K., Huggins, J.: A Toolset for Supporting UML Static and Dynamic Model Checking. In: Proc. 16th IEEE Int. Conf. Automated Software Engineering (ASE), pp. 147–152. IEEE, Los Alamitos (2001)

[SWK$^+$10] Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: Müller, W. (ed.) Proc. Design, Automation and Test in Europe (DATE 2010). IEEE, Los Alamitos (2010)

[SYC$^+$04] Sullivan, K.J., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA, pp. 133–142. ACM, New York (2004)

[TGF$^+$05] Trong, T.D., Ghosh, S., France, R.B., Hamilton, M., Wilkins, B.: UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In: Proc. OOPSLA Workshop on Eclipse Technology Exchange (Eclipse 2005), pp. 120–124. ACM Press, New York (2005)

# Proof Process Evaluation with Mutation Analysis
## Extended Abstract

Lydie du Bousquet and Michel Lévy

Laboratoire Informatique de Grenoble (LIG)
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{lydie.du-bousquet,michel.levy}@imag.fr

**Abstract.** In the context of deductive proof, formal specification (and thus proofs) may only cover a small part of the program behaviours. We discuss the relevance of applying a mutation analysis, proposed to evaluate the quality of a test set, to evaluate if a specification is able to detect predefined types of faults.

## 1 Introduction

A primary purpose of testing is to detect software failures. It implies running the software item in predetermined conditions (input selection), analyzing the obtained results, and identifying errors [16]. Testing can never completely establish the correctness of a program. For this reason, several methods (among which mutation analysis) have been put forward to increase confidence with respect to the test set provided.

Mutation analysis was introduced by DeMillo in 1978 [6]. Its main purpose is to evaluate the quality/adequacy of a test set. The basic idea is to insert changes into the program being tested, and to check if the tests are able to detect the difference between the original program and its variations. It is mainly used for unit testing evaluation. Since 1978, mutation analysis has been widespread, improved and evaluated. Is has been adapted for several programming languages [12]. Andrews *et al.* have demonstrated that this technique can provide a good indication of the fault detection ability of a test suite [3].

Testing is often opposed to verification techniques. In the following, we focus on theorem proving, which aims at assessing that an artefact (code or model) conforms to a (set of) property thanks to a sequence of deductions [19,17]. It is considered to be a reliable validation technique since it is based on well-founded mathematical deductions. However, when verifying an artefact with respect to a description (specifications), it is not necessary to describe all the facets of the artefact's behaviours. For instance, let us consider a program to sort a list. It can be specified that the resulting list should be sorted and should contain all the initial elements. But, one can forget to state that the size of the list should be unchanged. A code that adds new elements or removes some repeated values can be proved to be correct with respect to the description although it is not the expected behaviour.

In the following, we provide an example of how mutation analysis can help increase confidence with respect to the specification. Section 2 gives details about mutation analysis in the context of testing. Section 3 shows how mutation analysis can be used for proof. Section 4 considers some perspectives.

## 2 Mutation Analysis for Testing

Mutation analysis consists in introducing a small syntactic change in the source code of a program in order to produce a *mutant* [6] (for instance, replacing one operator by another or altering the value of a constant, etc.). Then the mutant behaviour is compared to the original program. If a difference can be *observed*, then the mutant is marked as *killed*. If the mutant has exactly the same observable behaviour as the original program, it is *equivalent*.

The original aim of the mutation analysis is the evaluation of a test set. To do that, one has to produce all mutants corresponding to a predefined fault model. If the test set can kill all non-equivalent mutants, the test set is declared *mutation-adequate*. This means that the tests are able to **discriminate the behaviours** of all the mutants from the original program. Mutation analysis does not confirm of the correctness of the program under test. It only focuses on the adequacy of the test data: the original program may be faulty and a mutant correct. If the tests are able to differentiate their behaviours, the tester may be able to detect the fault in the original program and correct it.

The adequacy of the test set is evaluated thanks to the *mutation score* (also called *adequacy score*). The mutation score is the percentage of non-equivalent mutants killed. For a program $P$, let $M_T$ be the total number of mutants produced with respect to a particular fault model $F$. Let $M_E$ and $M_K$ be the number of equivalent and killed mutants. The mutation score of the test set $T$ with respect to the fault model $F$ is defined as: $MS(P,T,F) = \frac{M_K}{M_T - M_E}$. A test set is *mutation-adequate* if the mutation score is equal to 1[1].

### Example: The `max` Program

Let us consider the following simple example of a program that computes the maximum of two values `x` and `y` (`max`).

```
int max(int x, int y) {if (x > y) return x; else return y;}
```

If one applies a fault model such as the one defined in [1] on the `max` program, one would obtain mutants in which `>` is replaced by `<` or `>=`; each instance of `x` is replaced by `y`, `x-1` or `x+1`; similarly each instance of `y` is replaced by `x`, `y-1` or `y+1`. For the sake of brevity, only three mutants are shown below.

[1] Mutation *testing* aims at producing tests until the maximal mutation score is obtained.

```
M1) int max(int x, int y) {if (x > y) return x; else return y+1;}
M2) int max(int x, int y) {if (x > y) return x; else return x;}
M3) int max(int x, int y) {if (x >= y) return x; else return y;}
```

`M1` was obtained by replacing `y` by `(y+1)` in the part `else` of the `if`. `M2` results from the replacement of variable `y` by `x` also in the `else` part of the `if`. `M3` is equivalent to the original program (`>` was replaced by `>=`).

For evaluating the mutation-adequacy of a test set, detecting the equivalent mutants is an important issue. Since, an equivalent mutant is impossible to kill, it is not possible to reach a mutation score equals to 1, if one is remaining. The program equivalence problem is undecidable, but several heuristics have been proposed to detect the equivalent mutants as much as possible [18].

## 3 Mutation Analysis for Proof

Let $P$ be a program and $S$ its specification. Once it has been established that $P$ satisfies $S$ ($P \models S$), one may want to know if the specification $S$ describes as many behaviours as expected. We propose to use mutation analysis in order to check if this specification is able to detect all faults defined by a fault model $F$.

Let us consider a mutant $M$ created from $P$ with respect to $F$. We say that a mutant $M$ is killed if $M$ does not satisfy $S$ ($M \not\models S$).
The mutation score of a *specification* $S$ with respect to the fault model $F$ is defined as previously: $MS(P,S,F) = \frac{M_K}{M_T - M_E}$.
A specification $S$ is *mutation-adequate* if $P \models S$ and $MS(P,S,F) = 1$.

### Example: The `max` Program

Let us specify the `max` program: (`S1`) "the result should be greater or equal to `x` and be greater or equals to `y`".

Let us prove the `max` program against this specification. We used the Why/Caduceus environment [10,11] to do that. "Why[2]" is a generic platform for deductive program verification [11]. Several provers can be used: proof assistants such as Coq, PVS, Simplify, etc. Programs to be proved have to be annotated in a language similar to JML (Java Modelling Language). For `S1`, we wrote:

```
/*@ ensures \result >= x && \result >= y @*/
```

Why/Caduceus produces 4 proof obligations for the program `max`. Those have been proved automatically in a few seconds by the different tools, so `Max` $\models$ `S1`.

We apply mutation analysis, to evaluate how precise our specification is, i.e. to evaluate how well the specification is able to discriminate the faulty programs. We obtain that

- `M1` $\models$ `S1`
- `M2` $\not\models$ `S1`
- `M3` $\models$ `S1`

[2] "Why" can be downloaded at `http://why.lri.fr/`

The Why/Caduceus environment manages to carry out all the proof-obligation for M3 since it is equivalent to the original program. The tools did not manage to carry out one proof-obligation for M2, which is unprovable[3]. This reveals that the specification is able to detect the fault introduced in M2. For M1, tools did not detect any contradiction with the specification. However, it is neither equivalent nor an acceptable version of the max. Here, the mutation-score of max specification is equal to 0.5 and has to be completed. To do that, we add one assertion, which specifies that the results of the program should be either equal to x or to y.

```
// S2
//@ ensures \result >= x && \result >= y && (\result == x || \result == y)
```

The original program is proved against the new specification. We obtain now

- M1 $\not\models$ S2[4],
- M2 $\not\models$ S2, and
- M3 $\models$ S2.

The new mutation-score is equal to 1. This allows us to strengthen our confidence with respect to the fact that the specification describes all the expected behaviours.

## 4 Perspectives

The example proposed here is a trial example. However, being able to evaluate if a specification describes as many behaviours as expected is an important issue. For several projects developed in our lab, we have noticed that the specifications proposed were correct but not detailed enough to detect as many faults as necessary. This occurred for several kinds of languages (JML, B , Lustre) or types of approaches (proof or model-checking) [7,13,8].

Mutation analysis has also been explored for evaluating specifications in the context of model-checking. For instance, in [20], the authors mutate a CSP specification for security analysis. Their main conclusion is that the equivalent specification provides an interesting source of information and helps to chose an appropriate alternative. In [15], the authors define several structural mutation

[3] The proof obligation max_impl_po_4 arises from the else part of the function. In this part, we know that `x <= y` (hypothesis H2) and we must ensure that the result has to verify `result >=y`. As the result is x, we must have `x >= y`.

[4] Six proof obligations are exhibited (instead of 4 previously). One of them is not proven by the tools. The proof obligation max_impl_po_6 comes from the else part of the function. In this part, we know that `x <= y` (hypothesis H2) and we must ensure that the result has to verify `(result=x or result=y)`. As the result is `y+1`, we must have `y+1=x` or `y+1=y`. This is unprovable, because `y+1=y` is false independently of hypothesis and under the hypothesis H2, `y+1=x` is false. This reveals the fault introduced.

models and coverage metrics to cover different design aspects in a state graph and to estimate the completeness of model checking.

The fact that there are more and more available environments to carry out verification[5] will increase the opportunities to prove our programs. Program verification may be carried out by persons who are less expert in proof than those who are currently proving programs. The need to evaluate the relevance of the specifications will then be more present, as we will need to evaluate the relevance of our tests. The use of mutation analysis can bring some new elements to evaluate a specification. Experimental work is required to evaluate this approach on real applications.

In the future, we would like to work on two points. The first one is related to the fault-model used for mutant production. Mutation analysis is an expensive approach. In [4], Budd estimates that the number of mutants can be approximated as the square of the number of statements (when applying a classic fault model). Since, most of the time, verification is a long and hard process, it is necessary to adjust the fault model in order to limit the production of mutants (especially equivalent mutants or mutants that will be trivially removed). This work requires an empirical evaluation of mutation operators in the context of proof, in order to select the most relevant ones. The hierarchy of fault class as defined in [14] can also be explored.

Second, specifications are often expressed as high-level properties. For instance, when specifying properties for a lift program [9], one may want to state that (1) all the requests to be served and that (2) never the doors open when the lift is moving. One should care to provide a specification that is not too restrictive, in order to accept several kinds of implementations. Producing a specification able to kill all mutants may result in an over-specification incompatible with implementation freedom. That is why it is important to be able to identify how and why a specification is modified. Modifying specification just to kill more mutants is probably not a good choice. That is why one should think about a process for the validation of specification in the context of proof.

*Acknowledgements.* This work was supported by the ANR project no. ANR-09-SEGI-014 (TASCCC).

## References

1. Agrawal, H., Demillo, R., Hathaway, R., Hsu, W.M., Hsu, W., Krauser, E., Martin, R.J., Mathur, A., Spafford, E.: Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Soft. Eng. Research Center, Dep. of Computer Science, Purdue Univ., Indiana (1989)
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4, 32–54 (2005)

[5] For Java, there are currently at least three available environments to carry out verification by proof: Why/Krakatoa, KeY [2] or JACK [5].

3. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, May 2005, pp. 402–411. ACM, New York (2005)
4. Budd, T.A.: Mutation analysis of program test data. Phd thesis, Yale University, New Haven, CT, USA (1980)
5. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: a developer-oriented approach. In: The 12th International FME Symposium, Pisa, Italy (September 2003)
6. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34–41 (1978)
7. du Bousquet, L.: Feature interaction detection using testing and model-checking, experience report. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 622–641. Springer, Heidelberg (1999)
8. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: Reusing a JML specification dedicated to verification for testing, and vice-versa: case studies. Journal of Automatic Reasoning (2009) (to appear)
9. Evans, A.S.: Specifying and Verifying Concurrent Systems Using Z. In: Naftalin, M., Bertrán, M., Denvir, T. (eds.) FME 1994. LNCS, vol. 873. Springer, Heidelberg (1994)
10. Filliâtre, J.-C., Marché, C.: Multi-prover verification of c programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
11. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
12. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. Technical Report TR-09-06, CREST Centre, King's College London, London, UK (2009)
13. Laleau, R., Ledru, Y., Bert, D., Bouquet, F., Lemoine, M., Dubois, C., Donzeau-Gouge, V., Vignes, S.: Using computer science modeling techniques for airport security certification. In: The 1st Int. Conf. on Research Challenges in Information Science (RCIS), pp. 61–72 (2007)
14. Lau, M.F., Yu, Y.T.: An extended fault class hierarchy for specification-based testing. ACM Trans. Softw. Eng. Methodol. 14(3), 247–276 (2005)
15. Lee, T.-C., Hsiung, P.-A.: Mutation coverage estimation for model checking. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 354–368. Springer, Heidelberg (2004)
16. Myers, G.: The Art of Software Testing. Wiley-Interscience, Hoboken (1979)
17. Newborn, M.: Automated Theorem Proving: Theory and Practice. Springer, Heidelberg (2001)
18. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Test., Verif. Reliab. 7(3), 165–192 (1997)
19. Rushby, J.M.: Theorem proving for verification. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 39–57. Springer, Heidelberg (2001)
20. Srivatanakul, T., Clark, J.A., Stepney, S., Polack, F.: Challenging formal specifications by mutation: a csp security example. In: The 10th Asia-Pacific Software Engineering Conference, USA, p. 340. IEEE, Los Alamitos (2003)

# Generating Regression Unit Tests Using a Combination of Verification and Capture & Replay

Christoph Gladisch[1], Shmuel Tyszberowicz[2], Bernhard Beckert[3], and Amiram Yehudai[4]

[1] Department of Computer Science, University of Koblenz
[2] School of Computer Science, The Academic College of Tel Aviv Yaffo
[3] Institut für Theoretische Informatik, Karlsruhe Institute of Technology
[4] School of Computer Science, Tel Aviv University

**Abstract.** The combination of software verification and testing techniques is increasingly encouraged due to their complementary strengths. Some verification tools have extensions for test case generation. These tests are strong at detecting software faults during the implementation and verification phase, and to further increase the confidence in the final software product. However, tests generated using verification technology alone may lack some of the benefits obtained by using more traditional testing techniques. One such technique is Capture and Replay, whose strengths are the generation of isolated unit tests and regression test oracles.

Hence, the two groups of techniques have complementary strengths, and therefore are ideal candidates for a tool-chain approach proposed in this paper. The first phase produces, for a given system, unit tests with high coverage. However, when using them to test a unit, its environment is tested as well – resulting in a high cost of testing. To solve this problem, the second phase captures the various executions of the program, which are monitored by the output of the first phase. The output of the second phase is a set of unit tests with high code coverage, which uses mock objects to test the units. Another advantage of this approach is the fact that the generated tests can also be used for regression testing.

## 1 Introduction

Formal verification is a powerful technique for ensuring functional correctness of software. Verification techniques that use symbolic execution and theorem proving, e.g. [2,5,4], can prove complex properties of a program when it is sufficiently annotated. However, failing verification attempts do not necessarily imply a fault in the program. In order to help the user in finding the reason for the failure, some verification tools have extensions for test case generation, e.g., [9,6,22]. Such verification-based testing (VBT) techniques use rich information about the program gained from the verification process. Furthermore, verification techniques based on model checking, e.g. [27,5], can also be regarded as VBT techniques. These techniques exhaustively enumerate the state space of a program and can detect faults in a program. These tools are highly automated but are typically bound to proving simpler program properties than techniques that use theorem provers.

Testing techniques, in contrast, are powerful for detecting software faults and for gaining some degree of confidence that the program under test (PUT) behaves correctly in its runtime environment. VBT techniques use information gained from a verification attempt and can generate very targeted tests to reveal program faults or tests that exhibit a high code coverage. Thus, both verification and testing techniques can profit when being combined. Yet, we can even go a step further in combining both approaches. We found that more traditional testing techniques have complementary strengths to VBT techniques. One such technique is capture and replay (CaR), whose strengths are the generation of isolated unit tests [20,21] and regression test oracles [20,28,8].

Unit testing plays a major role in the software development process. A unit test explores a particular behavior of the unit that is tested. The unit that we deal with is a class. It explores a particular aspect of the behavior of the class under test, hereafter CUT. Testing a unit in isolation is an important principle of unit testing [15]. However, the behavior of the CUT usually depends on other classes, some of them not even existing yet. *Mock objects* [19] are used to solve this problem by replacing actual calls to methods of other classes by calls that simply return the required value, thus testing the unit in isolation. Furthermore, in order to gain confidence in the test result the test should have a high code coverage.

The maintenance phase is the most expensive part of the software life cycle, and is estimated to comprise at least 50% of the total software development expenses [26]. Unit testing enables programmers to refactor code safely and make sure it works. Extreme Programming [31] adopts an approach that requires that *all* the software classes have unit tests; code without unit tests may not be released. Whenever code changes introduce a regression bug into a unit, it can quickly be identified and fixed. Hence, unit tests provide a safety net of regression tests and validation tests. This encourages developers to refactor working code, i.e., change its internal structure without altering the external behavior [12]. Research related to regression testing often focuses on test selection and test prioritization techniques, e.g. [14,16]. The focus of this paper is different. We exploit the *synergies* of combining VBT and CaR tools for unit regression testing.

We propose an approach for the automatic generation of unit and regression tests in the context of verification. Our goal is to improve test suites that are generated by VBT tools and CaR tools separately. The proposed approach maintains the high test coverage provided by VBT tools while at the same time reduces the complexity of the tests through automatic generation of mock objects. Using mock objects facilitates the isolation of the unit under test. Some existing CaR tools enable to create mock objects. On the other hand, CaR tools do not provide means to achieve high code coverage, and can therefore benefit from being combined with coverage guaranteeing tools such as VBT tools. The advantage of using VBT tools is that the verification process can be used to ensure that only correct behavior is captured by the CaR tool.

We identified that high code coverage and isolation are separate issues. They can be achieved independently using the two groups of techniques which have complementary strengths. Therefore we concluded that those groups of techniques are ideal candidates for the following tool-chain. The first phase produces, for a given system, unit tests with high code coverage. The second phase captures the various executions of the program, monitored by the output of the first phase. The output of the second phase is a set of unit tests with high coverage, which uses mock objects to test the units, in isolation.

The main contributions of the paper are described in Sections 2-4. We identify what the complementary strengths of VBT and CaR techniques are (Section 2). In Section 3 we present a novel tool-chain approach for unit regression testing in the context of verification and for unit regression testing in general. To the best of our knowledge, this tool-chain has not been considered with VBT tools so far. We have implemented the proposed approach using a concrete VBT and a concrete CaR tool resulting in the tool-chain KeYGenU. By applying KeYGenU to a small banking application we provide a proof of concept of our approach, as described in Section 4. The advantages and possible limitations of the approach are then discussed in Sections 3.2, 4.4, and 6. The other sections are related work (Section 5) and conclusions (Section 6).

## 2 Complementary Strengths of the Regarded Techniques

In the introduction we have described the complementary strengths of verification and testing in general. Both approachs should be combined in order to achieve reliable software and in order to optimize the verification and testing process. In this section we describe, by means of simple examples, advantages and disadvantages of CaR tools and coverage guaranteing tools like VBT tools that are more specific to our tool-chain approach.

*Regression Test Oracles.* Code that checks whether the result of a test-run is as expected is called *test oracle*. A *regression-test oracle* checks if the result is the same as in a previous version of the tested software.

Suppose there exists a well functioning application P. Let *evalExam(int points, int id)* be one of the methods of P returning a boolean value.

JAVA (2.1)

```java
public class Exam{
 boolean[] passed;
 public boolean evalExam(int points, int id){
  boolean res=false;
  if(points > 50){
    res=true;
  }
  passed[id] = res;
  return res;
}}
```

JAVA

Suppose that P has no regression test oracles and that P has been changed. Regression testing should be performed to avoid regression bugs. A CaR tool (e.g., [20,8]) can be used to create regression tests for the system. When executing `evalExam(40,2)`, for example, the CaR tool captures the return value of this method which is `false`. It then creates a unit test that executes `evalExam(40,2)` and compares the result with the previously observed value `false`. If, at the course of changes, the user mistakenly changes Line 4 to `res=true;`, the generated test will detect the bug as the return value is `true` and it differs from the previously captured return value `false`.

Assume now that the user enters a mistake in Line 6 rather than in Line 4, by changing Line 6 to `res=false;`. Then the generated unit tests do not detect the bug, because the execution of this branch was not captured.

*Code Coverage.* Using a VBT tool on the very same program produces a unit test suite with a high code coverage, i.e., a test is generated for both execution paths through `evalExam`. In order to create meaningful tests using the VBT tool, the user has to provide a requirement specification for `evalExam`. In our example we use the following JML requirement specification:

—— Java + JML (2.2) ——

```
/*@ public normal_behavior
    ensures \result==(points>50?true:false);@*/
public boolean evalExam(int points, int id){..}
```

—— Java + JML ——

Let us assume now that Line 4 has been changed to `res=true;` or that Line 6 has been changed to `res=false;`. In both cases the unit test suite generated by the VBT tool detects the bug.

By contrast, some CaR regression testing tools do not require writing a requirement specification, or even writing unit tests in advance, but there is a coverage problem with using CaR tools – unit tests are created only for the specific program run executed by the user or by a system test.

*Testing in Isolation.* Suppose the user changes the implementation of the method `evalExam()` by replacing the array `boolean[] passed` by a database management system. Line 8 is replaced by `passedDB.write(id,res);` that updates the database.

—— Java (2.3) ——

```
 3 public boolean evalExam(int points, int id){
 4   boolean res=false;
 5   if(points > 50){
 6     res=true;
 7   }
 8   passedDB.write(id,res);
 9   return res;
10 }
```

—— Java ——

The strength of VBT tools is the generation of test inputs that ensure a high test coverage. The tests, however, are not isolated unit tests because the execution of `evalExam` leads to the execution of `passedDB.write`.

Some existing CaR tools (e.g., [21,20]) can automatically create unit tests, using mock objects (see Section 3.1). This enables to perform unit testing in isolation, which in this case means that the generated unit test results in the execution of `evalExam` but not of `passedDB.write(id,res)`. Instead of calling the method `passedDB.write(id,res)` the generated mock object is activated which mimics a subset of input and output behavior of the database.

## 3 The Proposed Approach

We have analyzed the advantages and the problems of verification-based testing (VBT) tools and of capture and replay (CaR) tools separately. VBT tools support the verification process by helping to find software faults. They can generate test cases with high code coverage. These tools, however, usually generate neither mock objects nor regression test oracles that are based on previous program executions. CaR tools are strong at abstracting complicated program behavior and at automatically generating regression-test oracles. The CaR tools, however, can do this only for specific program runs, that have to be provided somehow. In contrast, VBT tools can generate program inputs for distinct program runs.

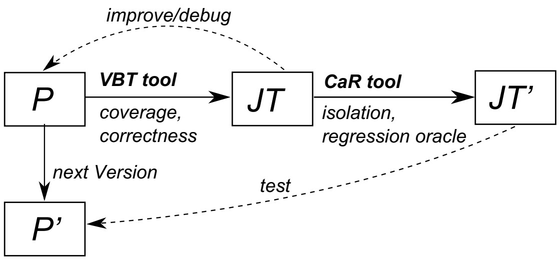


**Fig. 1.** The creation of a tool chain and its application to unit regression testing

From this analysis it becomes clear that these kinds of tools should be combined into a tool chain. Thus, the output of the VBT tool serves as input to the CaR tool, as shown in Figure 1. Our approach consists of two steps. In the first one the user tries to verify the program `P` using a verification tool that supports VBT. When a verification attempt fails, VBT is activated to generate a unit test suite `JT` for `P`. The so generated tests help in debugging `P` and the process is repeated until `P` is verifiable. When the verification succeeds the VBT tool is activated to generate a test suite `JT` that ensures coverage of the code of `P`. The generated test suite consists of one or more executable programs that are provided as input to the CaR tool. Thus when `JT` is executed the execution of the code under test is captured. The CaR tool in turn creates another unit test suite – `JT'`. If the CaR tool replays the observed execution of each test, consequently the high code coverage of `JT` is preserved by `JT'`. Furthermore, `JT'` benefits from the improvements that are gained by using the CaR tool. Depending on the capabilities of the CaR tool this can be the isolation of units and the extension of tests with regression-test oracles. Hence the tool chain employs the strengths of both kinds of tools involved. The test suite `JT'` can then be used to regression test `P'` that is the next development version of `P`.

## 3.1 Building a Tool-Chain

*Step I.* The goal of this step is to ensure the correctness of the code and to generate the test suite `JT` that ensure a high execution coverage. This can be achieved by using verification tools with their VBT extensions. In the following we describe such tools.

Bogor/Kiasan combines symbolic execution, model checking, theorem proving, and constraint solving to support design-by-contract reasoning of object-oriented software [5]. Its extension that we categorize as VBT is KUnit [6]. The tool focuses on heap-intensive JAVA programs and uses a lazy initialization algorithm with backtracking. The algorithm is capable of exploring all execution paths up to a bound on the configurations of the heap. KUnit then generates test data for each path and creates JUnit test suites. Similar features are provided by the KeY tool [2] that we describe in more detail in Section 4.1. ESC/Java2 [4] is a static checker that can automatically prove properties that go beyond simple assertions. A VBT extension of ESC/Java2 is Check'n'Crash [22]. It generates JUnit tests for assertions that could not be proved using ESC/Java2. In this way false warnings featured by ESC/Java2 are filtered out. This approach could be extended by providing unsatisfiable assertions that would stimulate Check'n'Crash to explore all execution paths of the PUT. Java PathFinder [27] is an explicit-state model checker. It is build on top of a custom-made Java Virtual Machine with nondeterministic choice and features the generation of test inputs. Thus it can be combined with a unit testing frame work like JUnit [17] to create `JT`.

*Step II.* The goal of the second step is to further improve the test suite `JT` using a CaR tool. When `JT` is executed, the CaR tool executes and captures each path through the method, generating `JT'`, a test suite for the PUT with the same coverage provided by `JT`. Depending on the used CaR tool, `JT'` may be a unit test suite supporting isolation or it may be extended with regression-test oracles.

In [21], test factoring is described that turns system tests into isolated unit tests by creating mock objects. For the capturing phase a wrapper class is created that records the program behavior to a transcript, and the replay step uses a mock class that reads from the transcript. The approach addresses complications that arise from field access, callbacks, object passing across boundaries, arrays, native method calls, and class loaders. The generation of mock objects is also supported by KUnit. The approaches, however, have different properties because in the latter approach mock objects are created from specifications instead of from runtime executions.

Some VBT tools can generate test oracles from the specifications that are used in the verification process. Such oracles are suitable for regression testing. Yet, not all parts of the system that are executed by `JT` may be specified. Our approach can be even applied if no test oracles are generated for `JT`. In this case a CaR tool like Orstra [28] can be used. During the capturing phase, Orstra collects object states to create assertions for asserting behavior of the object states. It also creates assertion that check return values of public methods with non-void returns. The assertions are then checked when the system is modified. In [8], a CaR approach is presented that creates regression tests from system tests. Components of the exercised system that may influence the behavior of the targeted unit are captured. A test harness is created that establishes the prestate of the unit that was encountered during system test execution. From that state, the unit is replayed and differences with the recorded unit poststate are detected.

GenUTest [20] is a CaR tool featuring both capabilities, i.e., the creation of isolated unit tests and the creation of regression-test oracles. It is described in Section 4.2.

## 3.2 Advantages and Limitations

We regard our approach from two perspectives. On the one hand, CaR tools can be used to further increase the quality of VBT. On the other hand, CaR tools can benefit from being combined with VBT tools. The VBT generated tests can be used to drive program's execution to ensure the coverage of the whole code. From this perspective our approach can be generalized by allowing general coverage ensuring tools for the first phase. However, for CaR tools, such as [8,28,20], it is important that during the capture phase only correct program behavior is observed – and this can be best ensured when a verification tool is used in the first phase.

The approach combines also the limitations of the involved tools. CaR-based regression testing tools can discover changes in the behavior when a program is modified, but they can not distinguish between intensional and not intensional changes. Another problem ocurrs with CaR tools that generate mock entities. It is often unclear under what preconditions the behavior of a mock entity is valid when the mock entity is executed in a state not previously observed by the CaR tool. Some advantages and limitations are specific to the particular tools and techniques. So are also the choice of the test target and mock objects. We advise the reader to refer to the referenced publications.

Verification tools are typically applicable to much smaller programs than testing tools. Our approach targets therefore at quality ensurance of small systems that are safety or security critical. Building a tool-chain adds complexity to the verification process. We expect, however, a payoff on the workload when the target system is modified and the quality of the software has to be maintained. Most VBT techniques are based on symbolic execution which is a challenging issue. Considering Listing 2.3 of Section 2, when symbolic execution reaches Line 8 the source code of `write()` may not be available or it may be too complicated for symbolic execution. Typically, in such situation method contracts that abstract the method call can be provided. Alternatively techniques such as [25] can be used that combine symbolic execution and runtime-execution.

Regression testing techniques such as [16], for example, are often concerned with test selection and test prioritization. The goal is to reduce the execution time of the regression test suite and thus to save costs. Graves et al. [14] describe test selection techniques for given regression test suites. They reduce the scope of the PUT that is executed by selecting a subset of the test suite. Our approach provides an alternative partitioning of the PUT (Figure 2) that can reduce its tested scope and should be considered in combination with test selection techniques. Instead of reducing the number of tests, parts of the program are substituted by mock entities.

When using selection techniques, a typical regression testing is usually described as follows (cf., for example, [14]). Let $P$ be the original version of the program, $P'$ the modified version that we would like to test, and $T$ is the test suite for $P$, then:

1. Select $T' \subseteq T$.
2. Test $P'$ with $T'$, establishing the correctness of $P'$ with respect to $T'$.
3. If necessary, create $T''$, a set of new functional or structural test cases for $P'$.

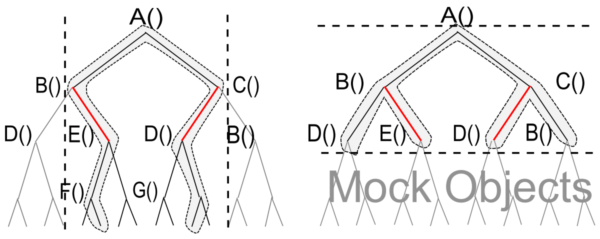


**Fig. 2.** The traditional test selection (left) versus our approach (right)

4. Test $P'$ with $T''$, establishing the correctness of $P'$ with respect to $T''$.
5. Create $T'''$, a new test suite and test execution profile for $P'$, from $T$, $T'$, and $T''$.

The authors of [14] point out the following problems associated with each of the steps:

1. It is not clear how to select a 'good' subset $T'$ of $T$ with which to test $P'$.
2. The problem of efficiently executing test suites and checking test results for correctness.
3. The coverage identification problem: the problem of identifying portions of $P'$ or its specification that require additional testing.
4. The problem of efficiently executing test suites and checking test results for correctness.
5. The test suite maintenance problem: the problem of updating and storing test information.

We use a slightly different model, which seems to solve the above issues. This model can be summarized as follows. Let $P$ be the original version of the program, $P'$ the modified version that we would like to test, and $T$ is the test suite which was generated for $P$ after running the proposed tool-chain.

1. Introducing mock objects produces $P'' \subseteq P'$.
2. Test $P''$ with $T$.
3. Rerun the tool-chain for the modified parts of $P'$ to produce $T'$, covering new branches.

The problems are solved as follows:

1. There is no need to select a subset $T'$ of $T$. Instead we have to consider how to create $P''$, i.e., which parts of the system $P'$ should be replaced by mock objects.
2. The problem of efficiently executing test suites and checking test results for correctness is solved by using mock objects, thus not executing the whole system.
3. The coverage identification problem is solved since the whole program may be tested.
4. Same as step 2.
5. The problem of updating and storing test information is solved by rerunning the tool-chain on the modified system parts.

Safe regression test selection techniques guarantee that the selected subset contains all test cases in the original test suite that can reveal regression bugs [14]. By executing only the unit tests of classes that have been modified a safe and simple selection technique should be obtained.

## 4 KeYGenU

We have implemented a concrete tool-chain according to Figure [1], called KeYGenU, and have applied it to several test cases. In this section we describe the two tools used by KeYGenU, namely KeY and GenUTest, and provide an example to demonstrate our ideas.

### 4.1 KeY

The KeY system [2] is a verification and test generation system for a subset of JAVA and a superset of JAVA CARD; the latter is a standardized subset of JAVA for programming of SmartCards. At its core, KeY is an automated and interactive theorem prover for first-order dynamic logic, a logic that combines first-order logic formulas with programs allowing to express, e.g., correctness properties of the programs.

KeY implements a VBT technique [10] with several extensions [9,13]. The test generation capabilities are based on the creation of a *proof tree* (see Figure [3]) for a formula expressing program correctness. The proof tree is created by interleaving first-order logic and symbolic execution rules where the latter execute the PUT with symbolic values in a manner that is similar to lazy evaluation. Case distinctions in the program are therefore reflected as branches of the proof tree; these may also be implicit distinctions like, e.g., the raising of exceptions. Proof tree branches corresponding to infeasible program paths, i.e., paths that can never be executed due to contradicting branch conditions in the program, are detected and not analyzed any further. Soundness of the system ensures that all paths through the PUT are analyzed, except for parts where the user chooses to use abstraction. Thus, creating tests for those proof branches often ensures full feasible path coverage of the regarded program part of the PUT. Based on the information contained in the proof tree, KeY creates test data using a built-in constraint solver. The PUT is initialized with the respective test data of each branch at a time. In this way execution of each program path in the proof tree is ensured.

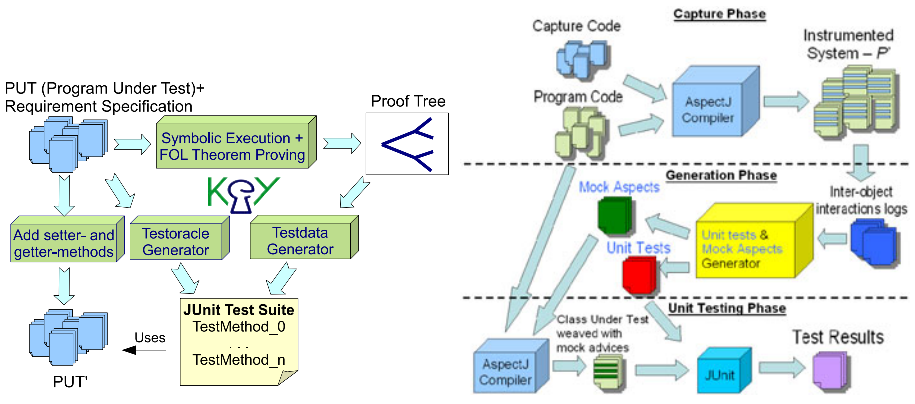


**Fig. 3.** Overview of verification-based testing implemented in KeY (left) and capture and replay implemented in GenUTest (right)

## 4.2 GenUTest

GenUTest is a prototype tool that generates unit tests [20]. The tool captures and logs inter-object interactions occurring during the execution of JAVA programs. The recorded interactions are then used to generate JUnit tests and mock-object like entities called mock aspects. These can be used independently by developers to test units in isolation. The comprehensiveness of the generated unit tests depends on the software execution. Software runs covering a high percentage generate in turn unit test with similar code coverage. Hence, GenUTest cannot guarantee a high coverage.

Figure 3 presents a high level view of GenUTest's architecture and highlights the steps in each of the three phases of GenUTest: the *capture phase*, the *generation phase*, and the *test phase*. In the capture phase the program is modified to include functionality to capture its execution. When the modified program executes, inter-object interactions are captured and logged. The interactions are captured by utilizing *AspectJ*, the most popular *Aspect-Oriented Programming* extension for the JAVA language [30,18]. The generation phase utilizes the log to generate unit tests and *mock aspects*, mock-object like entities. In the test phase, the unit tests are used by the developer to test the code of the program.

## 4.3 A Detailed Example

This section describes a simplified banking application, that was adopted from case studies on verification [3] and JML-based validation [7], and that was adapted to our needs. The bank customer can check his or her accounts as well as make money transfers between accounts. The customer can also set some rules for periodical money transfer. Figure 4 presents part of the case-study source-code.

The first step is to load the banking application into KeY and to select a method for symbolic execution; following the code excerpt in Figure 4, this is either `transfer()` or `registerSpendingRule()`. KeY generates a JUnit test suite from the obtained proof tree. It consists of a test method for every execution path of the method under test. Thus the test suite provides a high test coverage. Figure 5 shows one of the generated test methods for testing the method `transfer()`. In Lines 2–4 variables are declared and assigned initial values; Lines 5–9 assign test data to variables and fields; in Line 12 the method under test is executed; and in Line 16 the test oracle, implemented as `subformula5()`, is evaluated.

This test suite is the data that is exchanged from KeY to GenUTest. It is, however, a fully functioning test suite and should be executed before the continuation of the tool-chain, in order to automatically detect program bugs with respect to the JML-specification. In particular, this step turned out to be important because KeY is very good at detecting implicit program branches caused by, e.g., `NullPointerExceptions`, but on the other hand GenUTest expects the executed code *not* to throw any exception during capturing phase. Thus we have either extended the specifications, stating that certain fields are non-null, or we simply have removed from the test suite generated by KeY those test methods that have detected exceptions.

Capturing code of GenUTest is weaved-in into the KeY-generated test methods, such as in Figure 5, by running the test suite as an AspectJ application in the Eclipse

Java + JML

```
/* Copyright (c) 2002 GEMPLUS group. */
package banking; import ...;
public class Transfers_src {
 protected MyRuleVector rules=new MyRuleVector();
 private AccountMan_src accman;
 ... //field and method declarations

/*@ requires true;
modifies rules.size(), Rule.nbrules ;
ensures ((account <0 || spending_account <0)
          && (threshold > 0 && period >= 0))==> \result==3;
ensures (threshold<=0 && period>=0 && account>=0
                     && spending_account >=0)==> \result==5;
ensures (threshold>0 && period<0 && account>=0
                     && spending_account>=0) ==> \result==6;
...
signals (Exception e) false; @*/
public int registerSpendingRule(String date, int account, int threshold,
                                        int spending_account, int period) {
 if (account<0||spending_account<0)       return 3;
 Account account1 = accman.getRef(account);
 Account account2 = accman.getRef(spending_account);
 if ((account1==null)||(account2==null)) return 3;
 if (threshold <= 0)                      return 5;
 if (period < 0)                          return 6;
 Rule rule=new SpendingRule (date,account,
                     threshold,spending_account,period,accman);
 ...
}

/*@ requires true;
    ensures (amount<=0 ==> \result==1); @*/
public int transfer(int from_account, int to_account, int amount){
 Account fromAccount = accman.getRef(from_account);
 Account toAccount = accman.getRef(to_account);
 if(fromAccount!=null && toAccount!=null &&  amount > 0) {
  if(amount < fromAccount.getBalanceamount()){
     fromAccount.debit(amount);
     toAccount.credit(amount);
     return 0;
  }else
     return 1;
 }
 return 1;
} }//class declaration
```

Java + JML

**Fig. 4.** Excerpt from the banking case study

```java
public void  testcode0 () { /**declare vars**/
int from_account=0; int to_account=0; int res=0; int _to_account=0;
int _from_account=0; int _amount=0; int amount=0; Throwable exc=null;
Transfers_src o=null;
/**data**/ int testData0=2; int testData1=2;  o=new Transfers_src();
o._setrulesMyRuleVector(new MyRuleVector());
o._setaccmanAccountMan_src(new AccountMan_src());
from_account=testData0; to_account=testData1; _amnt=amount;
_from_account=from_account; _to_account=to_account;exc=null;

try { /** method under test **/
res=o.transfer(_from_account,_to_account,_amnt);
}  catch (java.lang.Throwable e) { exc=e; }

StringBuffer buffer=new StringBuffer();
boolean _oracleResult=subformula5(amount,exc,res,buffer);
assertTrue(buffer.toString(),_oracleResult);
}
```

**Fig. 5.** JUnit test method generated by KeY

IDE. After the capturing phase, GenUTest produces another JUnit test suite consisting of test methods like, e.g., in Figure 6, and mock aspects such as in Figure 7. As expected, the coverage of the KeY-generated tests is preserved by the GenUTest-generated tests; for instance, changes to any of the return values of the method `registerSpendingRule()` or the method `transfer()` have been detected.

Figure 6 presents the test method generated by GenUTest. The method invocations that were observed during the capture phase are replayed in Lines 4-14. GenUTest tries to minimize this code using some static analysis. The calls to `setSection()` are important for choosing the correct mock aspect as explained below. In Line 14 the actual method under test is called and its return value is compared in Line 15 with the value that was observed during capturing phase. Thus a regression test is performed.

In our experiments the calls to the methods `getRef()`, `getBalanceamount()`, `debit()`, and `credit()` (see Figure 4) were replaced, as expected, by mock aspect invocations, because these methods belong to classes different from the current class `Transfers_src`. For instance, Lines 2-4 in Figure 7 match the call to `getRef()` and Lines 7-11 check which occurrence of `getRef` in the call tree is currently processed, as different invocations may yield different return values. Line 11 checks if the given parameter value of `getRef()` has been actually observed during the capturing phase by using the reflection API. If this is not the case, then the original code is invoked with the current parameter value via the AspectJ keyword `proceed`, as shown in Line 11. Otherwise, the previously recorded return value is returned in Line 12, and thus unit testing in isolation is performed.

### 4.4 A Short Evaluation

We have tested KeYGenU on several use cases. It has automatically generated isolated unit-regression tests for classes of a banking application. Using the KeY-generated tests we have found several bugs in the application with respect to the provided JML-specification. This result confirms the observations made in [3,7] that the available specification was incomplete; e.g., many errors were caused by throwing `NullPointerExceptions` that should have been excluded by appropriate method preconditions. We have therefore either extended the specification or ignored these

JAVA
```java
@Test public void testtransfer1(){
 AccountMan_src AccountMan_src_11; MyRuleVector MyRuleVector_8;
 TestGeneric0 TestGeneric0_1; Transfers_src Transfers_src_4; int intRet;
 setSection("TestGeneric0",1,2);   TestGeneric0_1 = new TestGeneric0();
 setSection("Transfers_src",4,37); Transfers_src_4= new Transfers_src();
 setSection("MyRuleVector",40,67); MyRuleVector_8 = new MyRuleVector();
 setSection("Transfers_src",68,73);
 Transfers_src_4._setrulesMyRuleVector(MyRuleVector_8);
 setSection("AccountMan_src",76,129);
 AccountMan_src_11 = new AccountMan_src();
 setSection("Transfers_src",132,137);
 Transfers_src_4._setaccmanAccountMan_src(AccountMan_src_11);
 setSection("Transfers_src",140,149);
 intRetVal5 = Transfers_src_4.transfer(2,2,0);
 assertEquals(intRet,1);
 }
```
JAVA

**Fig. 6.** JUnit test method generated by GenUTest

AspectJ
```aspectj
pointcut restriction(): !adviceexecution() &&
   this(Transfers_src) && !target(Transfers_src);
Account around(int param1): call(banking.AccountMan_src.getRef(int))
                                    && args(param1) && restriction() {
 MockAspectHandler.Section currentSection =
        MockAspectHandler.getInstance().getClassSection("Transfers_src");
 if (currentSection.start == 884 && currentSection.end == 905){
  if (currentSection.statementCounter==1){
     currentSection.statementCounter++;
     Account Account_157 = new Account();
     if(reflectionCompare(param1,1)!=0){ return proceed(param1); }
     return Account_157;
  }}.../* commented out case distinctions */...}
```
AspectJ

**Fig. 7.** Mock aspect generated by GenUTest for the method `getRef()`

error-detecting test cases, as our focus was on regression testing. KeYGenU generated also unit tests for an old version of some software. Then, the unit tests have been executed with newer versions of the software. The discrepancies have been examined to determine if they uncover regression bugs. GenUTest generated a test suite that was able to detect changes to any branch of the tested methods, confirming the high test coverage.

Regarding scalability, KeYGenU generates in some cases a huge amount of unit tests. One of the reasons is that GenUTest generates tests not only for the method under test but also for the test code generated by KeY. For instance, the KeY-generated test oracle uses the class `StringBuffer` in order to collect debugging information about the evaluation of the post condition. This in turn resulted in over a hundred tests for the class `StringBuffer`. Also the selection of program paths is not optimized yet. Symbolic execution may lead to too many unwindings of loops producing many tests – some of which may be redundant, i.e., there may be more than one test that exercises the CUT in the same manner. These can be removed using the techniques described in [29].

## 5 Related Work

In Section 3.1 we described tools representing VBT techniques [9,6,22,27] as well as tools that represent CaR techniques [20,21,28,8]. In Section 3.2 we related our work to test selection and prioritization techniques [14,16]. Furthermore, a recent work that also automatically generates regression unit-tests is DiffGen [24]. In this approach the PUT is instrumented with additional branches and then a coverage-based test generation tool is used to detect regression bugs. In contrast, the approach presented in [23] suggests to use a verification tool for proving an equivalence relation between two version of a program. These approaches differ from ours as they do not use CaR techniques. In [28] the usage of a coverage guaranteeing tool is considered in combination with the CaR tool Orstra. However, the approaches used in [23,28] do not consider the generation of isolated unit tests and they do not provide means to guarantee that during capure phase the observed program behavior is correct.

Besides creating an approach for regression unit testing, our goal was also to investigate the combination of dynamic (runtime execution based) and static (symbolic execution based) analysis tools. Ernst [11] and Smaragdakis et al. [22] discuss the synergies and differences between static and dynamic analysis. The strength of static analysis is data generality and precision of code coverage, whereas the strength of dynamic analysis is speed of program execution and handling of black-box behavior without providing abstractions. While in [25], for example, static and dynamic analysis are combined in a rather coherent way, we suggest a tool-chain approach whose strength is the simplicity of the interface between the tools and their independence. Another tool-chain approach where KeY is used to obtain high code coverage has been realized in [1]. However, while in [1] a JML-specification is exchanged between the tools, in the here presented approach a unit test suite is exchanged from the VBT tool to the CaR tool.

## 6 Conclusion and Future Work

We have described an approach for automatic generation of unit tests that can also be used for regression testing. We aim at achieving high coverage of the tested code while testing each unit in isolation. This is accomplished by creating a tool-chain that combines two tools, a verification-based testing (VBT) and a capture and replay (CaR) test generation tool. We first run a VBT tool to generate tests for each path in a given system. This achieves a high coverage of the code, as desired. These tests are then used as input to a CaR tool that turns the tests into truly isolated unit tests by creating mock-object like entities. The advantage of using VBT tools is that the verification process can be used to ensure that only correct behavior is captured by the CaR tool.

To examine our ideas we have implemented KeYGenU, a concrete tool chain consisting of the VBT tool KeY and the CaR tool GenUTest. The tests that we have executed provide a proof of concept. The integration of different tools may, however, cause some additional work. For example, in the case of KeYGenU, the fact that both tools have been developed independently caused some difficulties. Running the tools in combination has revealed some bugs in each of the tools that have been fixed and that helped to improve both tools. GenUTest creates tests only for methods that return a value and only the returned value is analyzed by the generated regression tests. A considerable improvement would be to handle also void methods, e.g., by analyzing the state of the object on which the method was invoked.

Verification tools, such as KeY, are typically applicable to much smaller programs than testing tools. The scalability of the approach is bound by the scalability of the particular VBT and CaR tools. Our approach targets therefore at quality ensurance of small systems that are safety or security critical. Building the proposed tool-chain adds complexity to the verification process. The expected payoff on the workload is, however, when the target system is modified and the quality of the software has to be maintained.

**Acknowledgements.** We are grateful to Benny Pasternak for modifying GenUTest as needed to combine it with KeY. We also thank Jean-Louis Lanet for providing the banking application that served as our case study.

## References

1. Beckert, B., Gladisch, C.: White-box testing by combining deduction-based specification extraction and black-box testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 207–216. Springer, Heidelberg (2007)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: A developer-oriented approach. In: Int. Symp. Formal Methods Europe, pp. 422–439 (2003)
4. Cok, D.R., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
5. Deng, X., Robby, Hatcliff, J.: Kiasan: A verification and test-case generation framework for Java based on symbolic execution. In: ISoLA, Paphos, Cyprus, p. 137 (2006)
6. Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: TAICPART-MUTATION 2007: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, Washington, DC, USA, pp. 3–12. IEEE Computer Society Press, Los Alamitos (2007)
7. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: Case study in JML-based software validation. In: Proceedings, Automated Software Engineering, pp. 294–297 (2004)

8. Elbaum, S.G., Chin, H.N., Dwyer, M.B., Jorde, M.: Carving and replaying differential unit test cases from system test cases. IEEE Trans. Software Eng. 35(1), 29–45 (2009)
9. Engel, C., Gladisch, C., Klebanov, V., Rümmer, P.: Integrating verification and testing of object-oriented software. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 182–191. Springer, Heidelberg (2008)
10. Engel, C., Hähnle, R.: Generating Unit Tests from Formal Proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)
11. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: Workshop on Program Analysis For Software Tools and Engineering (PASTE), p. 35 (2004)
12. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
13. Gladisch, C.: Verification-based testing for full feasible branch coverage. In: Proceedings, Software Engineering and Formal Methods (SEFM). IEEE Computer Society Press, Los Alamitos (2008)
14. Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. TOSEM 10(2), 184–208 (2001)
15. Hamill, P.: Unit test frameworks. O'Reilly, Sebastopol (2004)
16. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for Java software. SIGPLAN Not. 36(11), 312–326 (2001)
17. Husted, T., Massol, V.: JUnit in Action. Manning Publications Co. (2003)
18. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning (2003)
19. Mackinnon, T., Freeman, S., Craig, P.: Endo-testing: unit testing with mock objects. In: Extreme Programming Examined, pp. 287–301. Addison-Wesley, Reading (2001)
20. Pasternak, B., Tyszberowicz, S., Yehudai, A.: GenUTest: a unit test and mock aspect generation tool. Journal on Software Tools for Technology Transfer (2009)
21. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for java. In: Proceedings, Automated Software Engineering, pp. 114–123 (2005)
22. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 1–16. Springer, Heidelberg (2007)
23. Strichman, O.: Regression verification: Proving the equivalence of similar programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, p. 63. Springer, Heidelberg (2009)
24. Taneja, K., Xie, T.: DiffGen: Automated regression unit-test generation. In: Proceedings, Automated Software Engineering (2008)
25. Tillmann, N., de Halleux, J.: Pex-white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
26. van Vliet, H.: Software Engineering: Principles and Practice, 2nd edn. John Wiley & Sons, Inc., Chichester (2000)
27. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11–14, pp. 97–107. ACM, New York (2004)
28. Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 380–403. Springer, Heidelberg (2006)
29. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proceedings, Automated Software Engineering, pp. 196–205 (2004)
30. AspectJ, http://www.eclipse.org/aspectj (Visited January 2010)
31. Extreme Programming, http://www.extremeprogramming.org (Visited January 2010)

# DyGen: Automatic Generation of High-Coverage Tests via Mining Gigabytes of Dynamic Traces

Suresh Thummalapenta[1], Jonathan de Halleux[2],
Nikolai Tillmann[2], and Scott Wadsworth[3]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC
sthumma@ncsu.edu
[2] Microsoft Research, One Microsoft Way, Redmond, WA
{jhalleux,nikolait}@microsoft.com
[3] Microsoft Corporation, One Microsoft Way, Redmond, WA
bwadswor@microsoft.com

**Abstract.** Unit tests of object-oriented code exercise particular sequences of method calls. A key problem when automatically generating unit tests that achieve high structural code coverage is the selection of relevant method-call sequences, since the number of potentially relevant sequences explodes with the number of methods. To address this issue, we propose a novel approach, called *DyGen*, that generates tests via mining dynamic traces recorded during program executions. Typical program executions tend to exercise only *happy* paths that do not include error-handling code, and thus recorded traces often do not achieve high structural coverage. To increase coverage, DyGen transforms traces into parameterized unit tests (PUTs) and uses dynamic symbolic execution to generate new unit tests for the PUTs that can achieve high structural code coverage. In this paper, we show an application of DyGen by automatically generating regression tests on a given version of software.[1]

**Keywords:** object-oriented unit testing, regression testing, dynamic symbolic execution.

## 1 Introduction

Software testing is a common methodology used to detect defects in the code under test. A major objective of unit testing is to achieve high structural coverage of the code under test, since unit tests can only uncover defects in those portions of the code, which are executed by those tests. Automatic generation of unit tests that achieve high structural coverage of object-oriented code requires method-call sequences (in short as *sequences*). These sequences help cover `true` or `false` branches in a method under test by creating desired object states for its receiver or arguments. We next present an example for desired object state and explain how method-call sequences help achieve desired object states using an illustrative example shown in Figure 1a.

Figure 1a shows an `AdjacencyGraph` class from the QuickGraph[2] library. The graph includes vertices and edges that can be added using the methods `AddVertex` and

[1] The majority of the work was done during an internship at Microsoft Research.
[2] `http://www.codeplex.com/quickgraph`

```
00:public class AdjacencyGraph {
01:    private VertexEdgesDictionary vertices;
02:    private EdgeCollection edges;
03:    public IVertex AddVertex() {
04:            IVertex v = Provider.ProvideVertex();
05:            vertices.Add(v);
06:            return v; }
07:    ...
08:    public IEdge AddEdge(IVertex source, IVertex target) {
09:            if (!vertices.ContainsKey(source))
10:                    throw new VertexNotFoundException("no vertex");
11:            if (!vertices.ContainsKey(target))
12:                    throw new VertexNotFoundException("no vertex");
13:            ...
14:            IEdge e = Provider.ProvideEdge(source,target);
15:            edges.Add(e);
16:            return e; }
17:    ...
18:    public void Compute() {
19:            if(edges.Count > 0) {
20:            ... }
22:} ... }
```

**a. AdjacencyGraph class from the QuickGraph library**

```
00:AdjacencyGraph g = new AdjacencyGraph();
01:Vertex s = g.AddVertex();
02:Vertex d = g.AddVertex();
03:Edge e = g.AddEdge(s, d);
04:g.Compute();
```

**b. A method-call sequence for generating a graph instance with an edge.**

```
00:void AddTest() {
01:            HashSet set = new HashSet();
02:            set.Add(7);
03:            set.Add(3);
04:            Assert.IsTrue(set.Count == 2);
05:}
```

**c. An example unit test**

```
00:void AddSpec(int x, int y) {
01:            HashSet set = new HashSet();
02:            set.Add(x);
03:            set.Add(y);
04:            Assert.AreEqual(x == y, set.Count == 1);
05:            Assert.AreEqual(x != y, set.Count == 2);
06:}
```

**d. An example PUT**

**Fig. 1.** Sample code examples

`AddEdge`, respectively. To reach Statement 20 in the `Compute` method, a desired object state is that the graph object should include at least one edge. Figure 1b shows a sequence that generates the desired object state. It is quite challenging to generate these sequences automatically from the implementation of `AdjacencyGraph` due to a large number of possible sequences and only a few sequences are valid. In practice, sequences required for generating desired object states often include multiple classes leading to a large space of possible sequences that cannot be effectively handled by existing approaches [1][2][3][4][5] that are either random or based on class implementations.

To address preceding issues, we propose a novel approach, called *DyGen*, that generates sequences from dynamic traces recorded during (typical) program executions. We use *dynamic* traces as opposed to *static* traces, since dynamic traces are more precise than static traces. These recorded dynamic traces include two aspects: realistic scenarios expressed as sequences and concrete values passed as arguments to those method calls. Since dynamic traces include both sequences and concrete argument values, these traces can directly be transformed into unit tests. However, such a naive transformation results in a large number of *redundant* unit tests that often do not achieve high structural coverage due to *two* major issues. We next explain these two major issues of naive transformation and describe how DyGen addresses those issues.

First, since dynamic traces are recorded during program executions, we identify that many of the recorded traces are duplicates. The reason for duplicates is that the same sequence can get invoked multiple times. Therefore, a naive transformation results in a large number of redundant unit tests. To address this issue, DyGen uses a combination of static and dynamic analyses and filters out duplicate traces.

Second, unit tests generated with the naive transformation tend to exercise only *happy paths* (such as paths that do not include error-handling code in the code under test) and often do not achieve high structural coverage of the code under test. To address this issue, DyGen transforms recorded dynamic traces into Parameterized Unit Tests (PUT) [6] rather than Conventional Unit Tests (CUT). PUTs are a recent advance in software testing and generalize CUTs by accepting parameters. Figure 1d shows a

PUT for the CUT shown in Figure 1c, where concrete values in Statements 2 and 3 are replaced by the parameters `x` and `y`. DyGen uses Dynamic Symbolic Execution (DSE) [7][8][9][10] to automatically generate a small set of CUTs that achieve high coverage of the code under test defined by the PUT. Section 2 provides more details on how DSE generates CUTs from PUTs. DyGen uses Pex [11], a DSE-based approach for generating CUTs from PUTs. However, DyGen is not specific to Pex and can be used with any other test-input generation engine.

DyGen addresses *two* major challenges faced by existing DSE-based approaches in effectively generating CUTs from PUTs. First, DSE-based approaches face a challenge in generating concrete values for parameters that require complex values such as floating point values or URLs. To address this challenge, DyGen uses naive transformation on each trace to generate a CUT, which is effectively an instantiation of the corresponding PUT. DyGen uses this CUT to seed the exploration of the corresponding PUT, which DyGen generates as well. Using seed tests helps not only to address the preceding challenge in generating complex concrete values, but also helps in increasing the efficiency of DSE while exploring PUTs. Second, in our evaluations (and also in practice), we identify that even after minimization of duplicate traces, the number of generated PUTs and seed tests can still be large, and it would take a long time (days or months) to explore those PUTs with DSE on a single machine. To address this challenge, DyGen uses a distributed setup that allows parallel exploration of PUTs.

In this paper, we show an application of DyGen by automatically generating regression tests on a given version of software. Regression testing, an important aspect of software maintenance, helps ensure that changes made in new versions of software do not introduce any new defects, referred to as *regression defects*, relative to the baseline functionality. Rosenblum and Weyuker [12] describe that the majority of software maintenance costs is spent on regression testing. To transform generated CUTs into regression tests, DyGen infers test assertions based on the given version of software. More specifically, DyGen executes generated CUTs on the given version of software, captures the return values of method calls, and generates test assertions from these captured return values. These test assertions help detect regression defects by checking whether the new version of software also returns the same values.

In summary, this paper makes the following major contributions:

– A scalable approach for automatically generating regression tests (that achieve high structural coverage of the code under test) via mining dynamic traces from program executions and without requiring any manual efforts.
– A technique to filter out duplicate dynamic traces by using static and dynamic analyses, respectively.
– A distributed setup to address scalability issues via parallel exploration of PUTs to generate CUTs.
– Three large-scale evaluations to show the effectiveness of our DyGen approach. In our evaluations, we show that DyGen recorded $\approx$1.5 GB C# source code (including 433,809 traces) of dynamic traces from applications using two core libraries of the .NET framework. From these PUTs, DyGen eventually generated 501,799 regression tests, where each test exercises a unique path, that together covered 27,485 basic blocks, which represents an increase of 24.3% over the number of blocks covered by the originally recorded dynamic traces.

## 2 Background

We next provide details of two major concepts used in the rest of the paper: dynamic symbolic execution and dynamic code coverage.

### 2.1 Dynamic Symbolic Execution

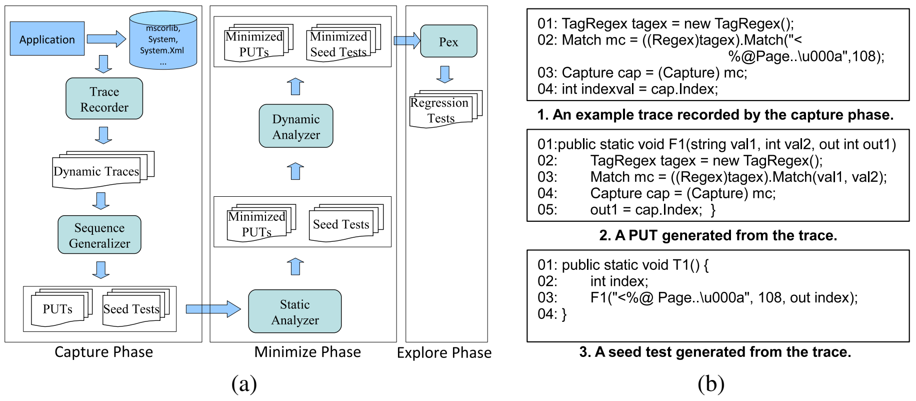
In our approach, we use Pex as an example state-of-the-art dynamic symbolic execution tool. Pex [11] is an automatic unit-test-generation tool developed by Microsoft Research. Pex accepts PUTs as input and generates CUTs that achieve high coverage of the code under test. Initially, Pex executes the code under test with arbitrary inputs. While executing the code under test, Pex collects constraints on inputs from predicates in branching statements along the exercised execution path. Pex next solves collected constraints to generate new inputs that guide future executions along new paths. Pex uses a constraint solver and theorem prover, called Z3 [13], to reason about collected constraints by faithfully encoding all constraints that arise in safe .NET programs. Z3 uses decision procedures for propositional logic, fixed sized bit-vectors, tuples, arrays, and quantifiers to reason about encoded constraints. Z3 approximates arithmetic constraints over floating point numbers by translating them to rational numbers. Pex also implements various optimization techniques to reduce the size of the formula that is given to Z3.

### 2.2 Dynamic Code Coverage

In this paper, we present dynamic code coverage information collected by Pex. As Pex performs code instrumentation dynamically at runtime, Pex only knows about the code that was already executed. In addition to code loaded from binaries on the disk, the .NET environment in which we perform our experiments allows the generation of additional code at runtime via *Reflection-Emit*.

## 3 Approach

Figure 2 shows the high-level overview of our DyGen approach. DyGen includes three major phases: *capture*, *minimize*, and *explore*. In the capture phase, DyGen records dynamic traces from (typical) program executions. DyGen next transforms these dynamic traces into PUTs and seed tests. Among recorded traces, we identify that there are many duplicate traces, since the same sequence of method calls can get invoked multiple times during program executions. Consequently, the generated PUTs and seed tests also include duplicates. For example, in our evaluations, we found that 84% of PUTs and 70% of seed tests are classified as duplicates by our minimize phase. To address this issue, in the minimize phase, DyGen uses a combination of static and dynamic analyses to filter out duplicate PUTs and seed tests, respectively. In the explore phase, DyGen uses Pex to explore PUTs to generate regression tests that achieve high coverage of the code under test.



```
01: TagRegex tagex = new TagRegex();
02: Match mc = ((Regex)tagex).Match("<
                                    %@Page..\u000a",108);
03: Capture cap = (Capture) mc;
04: int indexval = cap.Index;
```

```
01:public static void F1(string val1, int val2, out int out1)
02:       TagRegex tagex = new TagRegex();
03:       Match mc = ((Regex)tagex).Match(val1, val2);
04:       Capture cap = (Capture) mc;
05:       out1 = cap.Index;  }
```

```
01: public static void T1() {
02:       int index;
03:       F1("<%@ Page..\u000a", 108, out index);
04: }
```

**Fig. 2.** (a) A high-level overview of DyGen. (b) A dynamic trace and generated PUT and CUT from the trace.

## 3.1 Capture Phase

In the capture phase, DyGen records dynamic traces from program executions. The capture phase uses a profiler that records method calls invoked by the program during execution. The capture phase records both the method calls invoked and the concrete values passed as arguments to those method calls. Figure 2b1 shows an example dynamic trace recorded by the capture phase. Statement 2 shows the concrete value "`<% Page..\u000a`" passed as an argument for the `Match` method.

DyGen uses a technique similar to Saff et al. [14] for transforming recorded traces into PUTs and seed tests. To generate PUTs, DyGen identifies all constant values and promotes those constant values as parameters. Furthermore, DyGen identifies return values of method calls in the PUT and promotes those return values as `out` parameters for the PUT. In C#, these `out` parameters represent the return values of a method. DyGen next generates seed tests that include all concrete values from the dynamic traces. Figures 2b2 and 2b3 show the PUT and the seed test, respectively, generated from the dynamic trace shown in Figure 2b1.

The generated PUT includes two parameters and one `out` parameter. The `out` parameter is the return value of the method `Capture.Index`. These `out` parameters are later used to generate test assertions in regression tests (Section 3.3). The figure also shows a seed test generated from the dynamic trace. The seed test includes concrete values of the dynamic trace and invokes the generated PUT with those concrete values.

## 3.2 Minimize Phase

In the minimize phase, DyGen filters out duplicate PUTs and seed tests. The primary reason for filtering out duplicates is that exploration of duplicate PUTs or execution of duplicate seed tests is redundant and can also lead to scalability issues while generating regression tests. We use PUTs and seed tests shown in Figure 3 as illustrative examples to explain the minimize phase. The figure shows a method under test `foo`, two PUTs, and three seed tests. We use these examples primarily for explaining our minimize phase. Our actual PUTs are much more complex than these illustrative examples with

```
00:Class A {
01:     public void foo(int arg1, int arg2, int arg3) {
02:             if (arg1 > 0)
03:                 Console.WriteLine("arg1 > 0");
04:             else
05:                 Console.WriteLine("arg1 <= 0");
06:             if (arg2 > 0)
07:                 Console.WriteLine("arg2 > 0");
08:             else
09:                 Console.WriteLine("arg2 <= 0");
10:             for (int c = 1; c <= arg3; c++)  {
11:                     Console.WriteLine("loop");
12:             }
13:     }
14:}
```

```
00:void PUT1(int arg1, int arg2, int arg3) {
01:     A a = new A();
02:     a.foo(arg1, arg2, arg3); }

03:public void SeedTest1() {
04:     PUT1(1, 1, 1); }

05:void PUT2(int arg1, int arg2, int arg3) {
06:     A a = new A();
07:     a.foo(arg1, arg2, arg3); }

08:public void SeedTest2() {
09:     PUT2(1, 10, 1); }

10:public void SeedTest3() {
11:     PUT1(5, 8, 2); }
```

**Fig. 3.** Two PUTs and associated seed tests generated by the capture phase

an average PUT size of 21 method calls (Section 4.4). We first present our criteria for a duplicate PUT and a seed test and next explain how we filter out such duplicate PUTs and seed tests.

**Duplicate PUT:** We consider a PUT, say $P_1$, as a duplicate of another PUT, say $P_2$, if both $P_1$ and $P_2$ have the same sequence of Microsoft Intermediate Language (MSIL)[3] instructions.

**Duplicate Seed Test:** We consider a seed test, say $S_1$, as a duplicate of another seed test, say $S_2$, if both $S_1$ and $S_2$ exercise the same execution path. This execution path refers to the path that starts from beginning of the PUT that is called by the seed test, and goes through all (transitive) method calls performed by the PUT.

DyGen uses static analysis to identify duplicate PUTs. Consider the method bodies of `PUT1` and `PUT2`. DyGen considers `PUT2` as a duplicate of `PUT1`, since both the PUTs include the same sequence of MSIL instructions. Since `PUT2` is a duplicate of `PUT1`, DyGen automatically replaces the `PUT2` method call in `SeedTest2` with `PUT1`.

After eliminating duplicate PUTs, DyGen uses dynamic analysis for filtering out duplicate seed tests. To identify duplicate seed tests, DyGen executes each seed test and monitors its execution path in the code under test. For example, `SeedTest1` follows the path "3 $\rightarrow$ 7 $\rightarrow$ 11" in the `foo` method. DyGen considers `SeedTest2` as a duplicate of `SeedTest1`, since `SeedTest2` also follows the same path "3 $\rightarrow$ 7 $\rightarrow$ 11" in the `foo` method. Consider another unit test `SeedTest3` shown in Figure 3. DyGen does not consider `SeedTest3` as a duplicate of `SeedTest1`, since `SeedTest3` follows the path "3 $\rightarrow$ 7 $\rightarrow$ 11 $\rightarrow$ 11" (since `SeedTest3` iterates the loop in Statement 10 two times).

## 3.3 Explore Phase

In the explore phase, DyGen uses Pex to generate regression tests from PUTs. Although seed tests generated in the capture phase can be considered as regression tests, most seed tests tend to exercise common happy paths such as paths that do not include error-handling code in the code under test. In only a few rare scenarios, seed tests may exercise the paths related to error-handling code, if such scenarios happen during the recorded program executions. Therefore, these seed tests do not achieve high coverage of the corner cases and error handling of the code under test.

[3] `http://msdn.microsoft.com/en-us/library/c5tkafs1(VS.71).aspx`

```
00:[PexRaisedException(typeof(ArgumentNullEx))]
01:public static void F102() {
02:      int i = default(int);
03:      F1 ((string)null, 0, out i);
04: }
```

**a. Regression Test 1**

```
00:[PexRaisedException(typeof(ArgumentOutOfRangeEx))]
01:      public static void F110() {
02:            int i = default(int);
03:            F1("", 1, out i);
04: }
```

**b. Regression Test 2**

```
00: public static void F103() {
01:      int i = default(int);
02:      F1 ("\0\0\0\0\0\0\0<\u013b\0", 7, out i);
03:      PexAssert.AreEqual<int>(0, i);
04: }
```

**c. Regression Test 3**

**Fig. 4.** Regression tests generated by Pex by exploring the PUT shown in Figure 2b2

To address this issue, DyGen uses Pex to explore generated PUTs. Inspired by Patrice et al. [15], Pex can leverage seed inputs in the form of conventional unit tests. Using seed tests increases the effectiveness of Pex, and potentially any other DSE-based approaches, in two major ways. First, with seed tests, Pex executes those seed tests and internally builds an execution tree with nodes for all conditional control-flow statements executed along the paths exercised by the seed tests. Pex starts exploration from this pre-populated tree. In each subsequent iteration of the exploration, Pex tries to extend this tree as follows: a formula is constructed that represents the conjunction of the branch conditions of an already known path prefix, conjoined with the negation of a branch condition of a known suffix; the definitions of all derived values are expanded so that conditions only refer to the test inputs as variables. If the formula is satisfiable, and test inputs can be computed by the constraint solver, then by executing the PUT with those test inputs, Pex learns a new feasible path and extends the execution trees with nodes for the suffix of the new path. Without any seed tests, Pex starts exploration with an empty execution tree, and all nodes are discovered incrementally. Therefore, using seed tests significantly reduces the amount of time required in generating a variety of tests with potentially deep execution paths from PUTs. Second, seed tests can help cover reach certain paths that are hard to be covered without using those tests. For example, it is quite challenging for Pex or any other DSE-based approach to generate concrete values for variables that require complex values such as IP addresses, URLs, or floating point values. In such scenarios, seed tests can help provide desired concrete values to reach those paths.

Pex generated 86 regression tests for the PUT shown in Figure 2b2. Figure 4 shows three sample regression tests generated by Pex. In Regression tests 1 and 2, Pex automatically annotated the unit tests with expected exceptions `ArgumentNullException` and `ArgumentOutOfRangeException`, respectively. Since the PUT (Figure 2b2) includes an `out` parameter, Pex generated assertions in regression tests (such as Statement 3 in Regression test 3) based on actual values captured while generating the test. These expected exceptions or assertions serve as test oracles in regression tests.

When a PUT invokes code containing loops, an exhaustive exploration of all execution paths via DSE may not terminate. While Pex employs search strategies to achieve high code coverage quickly even in the presence of loops, Pex or any other DSE-based approaches may still take a long time (days or months) to explore PUTs with DSE on a single machine. To address this issue, DyGen uses an enhanced distributed setup originally proposed in our previous work [11]. Our distributed setup allows to launch multiple Pex processes on several machines. Once started, our distributed setup is designed to run forever in iterations. Each subsequent iterations increase bounds imposed on the exploration to guarantee termination. For example, consider the *timeout* parameter that describes when to stop exploring a PUT. In the first iteration, DyGen sets three minutes for the timeout parameter. This value indicates that DyGen terminates exploration of a PUT after three minutes. In the first iteration, DyGen explores all PUTs with these bounded parameters. In the second iteration, DyGen doubles the values of these parameters. For example, DyGen sets six minutes for the timeout parameter in the second iteration. Doubling the parameters gives more time for Pex in exploring new paths in the code under test. To avoid Pex exploring the same paths that were explored in previous iterations, DyGen maintains a pool of all generated tests. DyGen uses the tests in the pool generated by previous iterations as seed tests for further iterations. For example, tests generated in Iteration 1 are used as seed tests in Iteration 2. Based on the amount of time available for generating tests, tests can be generated in further iterations.

## 4 Evaluations

We conducted three evaluations to show the effectiveness of DyGen in generating regression tests that achieve high coverage of the code under test. Our empirical results show that DyGen is scalable and can automatically generate regression tests for large real-world code bases without any manual efforts. In our evaluations, we use two core .NET 2.0 framework libraries[4] as main subjects. We next describe the research questions addressed in our evaluation and present our evaluation results.

### 4.1 Research Questions

We address the following three research questions in our evaluations.

- RQ1: Can DyGen handle large real-world code bases in automatically generating regression tests that achieve high coverage of the code under test?
- RQ2: Do seed tests help achieve higher coverage of the code under test than without using seed tests?
- RQ3: Can more machine power help generate new regression tests that can achieve more coverage of the code under test?

### 4.2 Subject Code Bases

We used two core .NET 2.0 framework base class libraries as the main subjects in our evaluations. Since these libraries sit at the core of the .NET framework, it is paramount for the .NET product group to maintain and continually enrich a comprehensive regression test suite, in order to ensure that future product versions preserve the existing behavior, and to detect breaking changes. Table 1 shows the two libraries (mscorlib

[4] `http://msdn.microsoft.com/en-us/library/ms229335.aspx`

**Table 1.** Ten .NET framework base class libraries used in our evaluations

| .NET libraries | Short name | KLOC | # public classes | # public methods |
|---|---|---|---|---|
| mscorlib | mscorlib | 178 | 1316 | 13199 |
| System | System | 149 | 947 | 8458 |
| System.Windows.Forms | Forms | 226 | 1403 | 17785 |
| System.Drawing | Drawing | 24 | 223 | 2823 |
| System.Xml | Xml | 122 | 270 | 5426 |
| System.Web.RegularExpressions | RegEx | 10 | 16 | 162 |
| System.Configuration | Config | 17 | 105 | 773 |
| System.Data | Data | 126 | 298 | 5464 |
| System.Web | Web | 202 | 1140 | 11487 |
| System.Transactions | Trans | 9.5 | 39 | 405 |
| **TOTAL** | | 1063 | 5757 | 65982 |

and System) used in our evaluations and their characteristics such as the number of classes and methods. Column "Short name" shows short names (for each library) that are used to refer to those libraries. The table also shows statistics of eight other libraries of .NET 2.0 framework. Although these other eight libraries are not our primary targets for generating regression tests, they were exercised as well by the recorded program executions. In our evaluations, we use these additional eight libraries also while presenting our coverage results. The table shows that these libraries include 1,063 KLOC with 5,757 classes and 65,982 methods.

## 4.3 Evaluation Setup

In our evaluations, we used nine machines that can be classified into three configuration categories. On each machine, we launched multiple Pex processes. The number of processes launched on a machine is based on the configuration of the machine. For example, on an eight core machine, we launched seven Pex processes. Each Pex process was exploring one class (including multiple PUTs) at a time. Table 5(a) shows all three configuration categories. Columns "# of mc" and "# of pr" show the number of machines of each configuration and the number of Pex processes launched on each machine, respectively.

Since we used .NET framework base class libraries in our evaluations, the generated tests may invoke method calls that can cause external side effects and change the machine configuration. Therefore, while executing the code during exploration of PUTs or while running generated tests, we created a sand-box with the "Internet" security permission. This permission represents the default policy permission set for the content from an unknown origin. This permission blocks all operations that involve environment interactions such as file creations or registry accesses by throwing `SecurityException`. We adopted sand-boxing after some of the Pex generated tests had corrupted our test machines. Since we use a sand-box in our evaluations, the reported coverage is lower than the actual coverage that can be achieved by our generated regression tests.

To address our research questions, we first created a base line in terms of the code coverage achieved by the seed tests, referred to as *base coverage*. In our evaluations, we use block coverage (Section 2.2) as a coverage criteria. We report our coverage in terms

| Machine Configuration | # of mc | # of pr |
|---|---|---|
| Xeon 2 CPU @ 2.50 GHz, 8 cores, 16 GB RAM | 1 | 7 |
| Quad core 2 CPU @ 1.90 GHz, 8 cores, 8 GB RAM | 2 | 7 |
| Intel Xeon CPU @2.40 GHz, 2 cores, 1 GB RAM | 6 | 1 |

(a)

| Mode | # of Tests | # of blocks | % of incr from base |
|---|---|---|---|
| WithoutSeeds Iteration 1 | 248,306 | 21,920 | 0% |
| WithoutSeeds Iteration 2 | 412,928 | 23,176 | 4.8% |
| WithSeeds Iteration 1 | 376,367 | 26,939 | 21.8% |
| WithSeeds Iteration 2 | 501,799 | 27,485 | 24.3% |

(b)

**Fig. 5.** (a) Three categories of machine configurations used in our evaluations. (b) Generated regression tests.

of the number of blocks covered in the code under test. We give only an approximate upper bound on the number of reachable basic blocks, since we do not know which blocks are actually reachable from the given PUTs for several reasons: we are executing the code in a sand-box, existing code is loaded from the disk only when it is used and new code may be generated at runtime.

We next generated regression tests in four different modes. In Mode "*WithoutSeeds Iteration 1*", we generated regression tests without using seed tests for one iteration. In Mode "*WithoutSeeds Iteration 2*", we generated regression tests without using seed tests for two iterations. The regression tests generated in Mode "WithoutSeeds Iteration 2" are a super set of the regression tests generated in Mode "WithoutSeeds Iteration 1". In Mode "*WithSeeds Iteration 1*", we generated regression tests with using seed tests for one iteration. Finally, in Mode "*WithSeeds Iteration 2*", we generated regression tests with using seed tests for two iterations. Modes "WithoutSeeds Iteration 1" and "WithSeeds Iteration 1" took one and half day for generating tests, whereas Modes "WithoutSeeds Iteration 2" and "WithSeeds Iteration 2" took nearly three days, since these modes correspond to Iteration 2.

## 4.4 RQ1: Generated Regression Tests

We next address the first research question of whether DyGen can handle large real-world code bases in automatically generating regression tests. This research question helps show that DyGen can be used in practice and can address scalability issues in generating regression tests for large code bases. We first present the statistics after each phase in DyGen and next present the number of regression tests generated in each mode.

In the capture phase, DyGen recorded 433,809 dynamic traces and persisted them as C# source code, resulting in ≈1.5 GB of C# source code. The average trace length includes 21 method calls and the maximum trace length includes 52 method calls. Since our capture phase transforms each dynamic trace into a PUT and a seed test, the capture phase resulted in 433,809 PUTs and 433,809 seed tests.

In the minimize phase, DyGen uses static analysis to filter out duplicate PUTs. Our static analysis took 45 minutes and resulted in 68,575 unique PUTs. DyGen uses dynamic analysis to filter out duplicate seed tests. Our dynamic analysis took 5 hours and

**Table 2.** Comparison of coverage achieved for ten .NET libraries used in our evaluation

| **.NET libraries** | **Maximum Coverage** | **Base Coverage** | **WithOutSeeds Iteration 1** | | **WithOutSeeds Iteration 2** | | **WithSeeds Iteration 1** | | **WithSeeds Iteration 2** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # blocks | # blocks | # blocks | % increase | # blocks | % increase | # blocks | % increase | # blocks | % increase |
| mscorlib | 20437 | 12827 | 13063 | 1.84 | 13620 | 6.18 | 14808 | 15.44 | 15018 | 17.08 |
| System | 7786 | 4651 | 4062 | -12.67 | 4243 | -8.77 | 5907 | 27.00 | 6039 | 29.84 |
| Forms | 2815 | 1730 | 1572 | -9.13 | 1774 | 2.54 | 1782 | 3.01 | 1865 | 7.80 |
| Drawing | 850 | 570 | 580 | 1.75 | 591 | 3.68 | 618 | 8.42 | 625 | 9.65 |
| Xml | 2770 | 1229 | 1390 | 13.10 | 1462 | 18.96 | 1959 | 59.40 | 2045 | 66.40 |
| RegEx | 854 | 351 | 330 | -5.98 | 520 | 48.15 | 754 | 114.81 | 771 | 119.66 |
| Config | 392 | 263 | 297 | 12.93 | 297 | 12.93 | 302 | 14.83 | 306 | 16.35 |
| Data | 865 | 301 | 380 | 26.25 | 422 | 40.20 | 562 | 86.71 | 569 | 89.04 |
| Web | 253 | 154 | 211 | 37.01 | 212 | 37.66 | 212 | 37.66 | 212 | 37.66 |
| Trans | 59 | 35 | 35 | 0.00 | 35 | 0.00 | 35 | 0.00 | 35 | 0.00 |
| **TOTAL/AVG** | **37081** | **22111** | **21920** | **$<0$** | **23176** | **4.80** | **26939** | **21.80** | **27485** | **24.30** |

resulted in 128,185 unique seed tests. These results show that there are a large number of duplicate PUTs and seed tests, and show the significance of our minimize phase. We next measured the block coverage achieved by these 128,185 unique seed tests in the code under test and used this coverage as *base coverage*. These tests covered 22,111 blocks in the code under test.

Table 5(b) shows the number of regression tests generated in each mode along with the number of covered blocks. The table also shows the percentage of increase in the number of blocks compared to the base coverage. As shown in results, in Mode "WithSeeds Iteration 2", DyGen achieved 24.3% higher coverage than the base coverage. Table 2 shows more detailed results of coverage achieved for all ten .NET libraries. Column ".NET libraries" shows libraries under test. Column "Maximum Coverage" shows an approximation of the upper bound (in terms of number of blocks) of achievable coverage in each library under test. In particular, this column shows the sum of all blocks in all methods that are (partly) covered by any generated test. However, we do not present the coverage results of our four modes as percentages relative to these upper bounds, since these upper bounds are only approximate values, whereas the relative increase of achieved coverage can be measured precisely. Column "Base Coverage" shows the number of blocks covered by seed tests for each library. Column "WithOutSeeds Iteration 1" shows the number of blocks covered ("# blocks") and the percentage of increase in the coverage ("% increase") with respect to the base coverage in this mode. Similarly, Columns "WithOutSeeds Iteration 2", "WithSeeds Iteration 1", and "WithSeeds Iteration 2" show the results for the other three modes.

Since we use seed tests during our exploration in Modes "WithSeeds Iteration 1" or "WithSeeds Iteration 2", the coverage achieved is either the same or higher than the base coverage. However, DyGen has achieved significant higher coverage than base coverage for libraries mscorlib and System (in terms of the number of additional blocks covered). The primary reason is that most of the classes in these libraries are stateless and do not require environment interactions. The results show that DyGen can handle large real-world code bases and can generate large number of regression tests that achieve high coverage of the code under test.

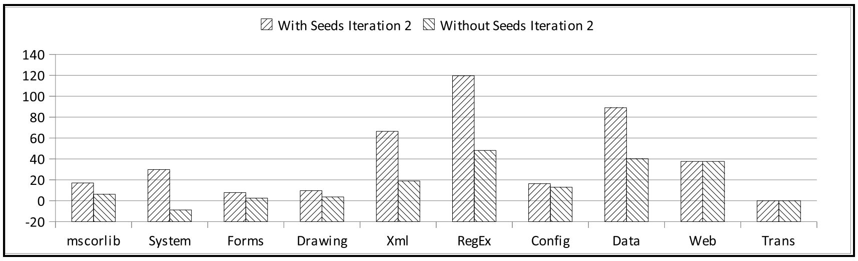


**Fig. 6.** Comparison of coverage achieved by Mode “WithSeeds Iteration 2” and Mode “WithoutSeeds Iteration 2”

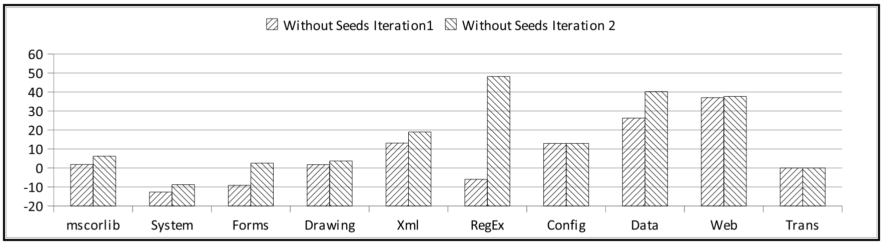


**Fig. 7.** Comparison of code coverage achieved by Modes “WithoutSeeds Iteration 1” and “WithoutSeeds Iteration 2”

### 4.5 RQ2: Using Seed Tests

We next address the second research question of whether seed tests help achieve higher code coverage compared to without using seed tests. To address this question, we compare the coverage achieved by generated tests in Modes “WithoutSeeds Iteration 2” and “WithSeeds Iteration 2”. Figure 6 shows comparison of the coverage achieved in these two modes. The x-axis shows the library under test and y-axis shows the percentage of increase in the coverage with respect to the base coverage. As shown, Mode “WithSeeds Iteration 2” always achieved higher coverage than Mode “WithoutSeeds Iteration 2”. On average “WithSeeds Iteration 2” achieved 18.6% higher coverage than “WithoutSeeds Iteration 2”. The table also shows that there is a significant increase in the coverage achieved for the `System.Web.RegularExpressions` (`RegEx`) library. In Section 3.3, we described one of the major advantages of seed tests is that seed tests can help cover certain paths that are hard to be covered without using those tests. The `System.Web.RegularExpressions` library is an example for such paths since this library requires complex regular expressions to cover certain paths in the library. It is quite challenging for Pex or any other DSE-based approach to generate concrete values that represent regular expressions. The increase in the coverage for this library shows that concrete values in the seed tests help achieve higher coverage. In summary, the results show that seed tests help achieve higher coverage compared to without using seed tests.

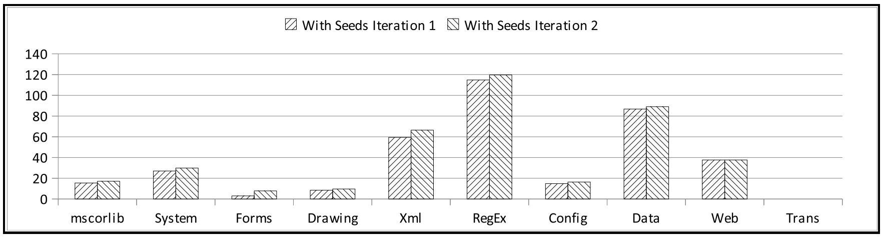


**Fig. 8.** Comparison of code coverage achieved by Modes "WithSeeds Iteration 1" and "Withseeds Iteration 2"

## 4.6 RQ3: Using More Machine Power

We next address the third research question of whether more machine power helps achieve more coverage. This research question helps show that additional coverage can be achieved in further iterations of DyGen. To address this question, we compare coverage achieved in Mode "WithoutSeeds Iteration 1" with Mode "WithoutSeeds Iteration 2", and Mode "WithSeeds Iteration 1" with Mode "WithSeeds Iteration 2" (shown in Table 2).

Figure 7 shows the comparison of coverage achieved in Modes "WithoutSeeds Iteration 1" and "WithoutSeeds Iteration 2". On average, Mode "WithoutSeeds Iteration 2" achieved 5.73% higher coverage than Mode "WithoutSeeds Iteration 1". This result shows that DyGen can achieve additional coverage in further iterations. However, the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 1" is not doubled. The primary reason is that it gets harder to cover new blocks in further iterations.

Figure 8 shows the comparison of coverage achieved in Modes "WithSeeds Iteration 1" and "WithSeeds Iteration 2". On average, Mode "WithSeeds Iteration 2" achieved 2.0% higher coverage than Mode "WithSeeds Iteration 1". The increase in coverage from Mode "WithSeeds Iteration 1" to Mode "WithSeeds Iteration 2" is less than the increase in the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 2". This difference is due to seed tests that help achieve higher coverage during Mode "WithSeeds Iteration 1", leaving more harder blocks to be covered in Mode "WithSeeds Iteration 2". In summary, the results show that further iterations can help generate new regression tests that can achieve more coverage.

## 5 Discussion and Future Work

Although our generated tests achieved higher coverage (24.3%) than the seed tests, we did not achieve full overall coverage of our subject code bases (i.e. 100% coverage of all methods stored in the code bases on disk). There are three major reasons for not achieving full coverage. First, using a sand-box reduces the amount of executable code. Second, our recorded dynamic traces do not invoke all public methods of the libraries under analyses. In future work, we plan to address this issue by generating PUTs for all public methods that are not covered. Third, the code under test includes branches that cannot be covered with the test scenarios recorded during program executions. To address this issue, we plan to generate new test scenarios from existing scenarios by using evolutionary techniques [2].

We did not find any previously unknown defects while generating regression tests. We did not expect to find defects, since our subject code bases are well tested both manually and by automated tools, including research tools such as Randoop [5] and Pex [11]. Although regression testing is our ultimate goal, in our current approach, we primarily focused on generating regression tests that achieve high code coverage of the given version of software. In future work, we plan to apply these regression tests on further versions of software in order to detect regression defects. Furthermore, in our evaluation, we used two libraries as subject applications. However, our approach is not specific for libraries and can be applied to any application in practice.

## 6 Related Work

Our approach is closely related to two major research areas: regression testing and method-call sequence generation.

**Regression testing.** There exist approaches [16][17][14] that use a capture-and-replay strategy for generating regression tests. In the capture phase, these approaches monitor the methods called during program execution and use these method calls in the replay phase to generate unit tests. Our approach also uses a strategy similar to the capture-and-replay strategy, where we capture dynamic traces during program execution and use those traces for generating regression tests. However, unlike existing approaches that replay exactly the same captured behavior, our approach replays beyond the captured behavior by using DSE in generating new regression tests.

Another existing approach, called Orstra [18], augments an existing test suite with additional assertions to detect regression faults. To add these additional assertions, Orstra executes a given test suite and collects the return values and receiver object states after the execution of methods under test. Orstra generates additional assertions based on the collected return values or receiver object states. Our approach also uses a similar strategy for generating assertions in the regression tests. Another category of existing approaches [19][20][21] in regression testing primarily target at using regression tests for effectively exposing the behavioral differences between two versions of a software. For example, these approaches target at selecting those regression tests that are relevant to portions of the code changed between the two versions of software. However, all these approaches require an existing regression test suite, which is the primary focus of our current approach.

**Method-call sequence generation.** To test object-oriented programs, existing test-generation approaches [3][4][22][23] accept a class under test and generate sequences of method calls randomly. These approaches generate random values for arguments of those method calls. Another set of approaches [1] replaces concrete values for method arguments with symbolic values and exploits DSE techniques [7][8][9][10] to regenerate concrete values based on branching conditions in the method under test. However, all these approaches cannot handle multiple classes and their methods due to a large search space of possible sequences.

Randoop [5] is a random testing approach that uses an incremental approach for constructing method-call sequences. Randoop randomly selects a method call and finds arguments required for these method calls. Randoop uses previously constructed method-call sequences to generate arguments for the newly selected method call. Randoop may also pick values for certain primitive randomly, or from a fixed manually supplied pool of values. Randoop incorporates feedback obtained from previously constructed method-call sequences while generating new sequences. As soon as a method-call sequence is constructed, Randoop executes the sequence and verifies whether the sequence violates any contracts and filters. Since Randoop does not symbolically analyze how the code under test uses arguments, Randoop is often unable to cover data-dependent code paths. On the other hand, DyGen is dependent on method-call sequences obtained via dynamic traces, and so DyGen is often unable to cover code paths that cannot be covered from the scenarios described by those sequences. Therefore, Randoop and DyGen are techniques with orthogonal goals and effects. In our previous approach [11], we applied Pex on a core .NET component for detecting defects. Unlike our new approach that uses realistic scenarios recorded during program executions, our previous approach generates individual PUTs for each public method of all public classes. There, we could not cover portions of the code that require long scenarios. Our new approach complements our previous approach by using realistic scenarios for covering such code portions.

Our approach is also related to another category of approaches based on mining source code [24][25] [26]. These approaches statically analyze code bases and use mining algorithms for extracting frequent patterns. These frequent patterns are treated as programming rules in either assisting programmers while writing code or for detecting violations as deviations from these patterns. Unlike these existing approaches, our approach mines dynamic traces recorded during program executions and uses those traces for generating regression tests. Our previous work [26] also mines method-call sequences from existing code bases. Our previous work uses these method-call sequences to assist random or DSE-based approaches. Our new approach is significantly different from our previous work in three major aspects. First, our new approach is a complete approach for automatically generating regression tests from dynamic traces, whereas, our previous work mines method-call sequences to assist random or DSE-based approaches. Second, our new approach uses dynamic traces, which are more precise compared to the static traces used in our previous work. Third, our new approach includes additional techniques such as seed tests and distributed setup for assisting DSE-based approaches in effectively generating CUTs from PUTs.

## 7 Conclusion

Automatic generation of method-call sequences that help achieve high structural coverage of object-oriented code is an important and yet a challenging problem in software testing. Unlike existing approaches that generate sequences randomly or based on analysis of the methods, we proposed a novel scalable approach that generates sequences via mining dynamic traces recorded during (typical) program executions. In this paper, we showed an application of our approach by automatically generating regression tests for two core .NET 2.0 framework libraries. In our evaluations, we showed that our approach recorded ≈1.5 GB (size of corresponding C# source code) of dynamic traces and eventually generated ≈500,000 regression tests, where each test exercised a unique path. The generated regression tests covered 27,485 basic blocks, which represents an improvements of 24.3% over the number of blocks covered by the original recorded dynamic traces. These numbers show that our approach is highly scalable and can be used in practice to deal with large real-world code bases. In future work, we plan to evaluate the effectiveness of generated regression tests in detecting behavioral differences between two versions of software.

## References

1. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 297–306. IEEE Computer Society Press, Los Alamitos (2008)
2. Tonella, P.: Evolutionary testing of classes. SIGSOFT Softw. Eng. Notes 29(4), 119–128 (2004)
3. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Softw. Pract. Exper. 34(11), 1025–1050 (2004)
4. Parasoft: Jtest manuals version 5.1. Online manual (2006), http://www.parasoft.com
5. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering, Minneapolis, MN, USA, pp. 75–84. IEEE Computer Society, Los Alamitos (2007)
6. Tillmann, N., Schulte, W.: Parameterized unit tests. SIGSOFT Softw. Eng. Notes 30(5), 253–262 (2005)
7. Clarke, L.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. 2(3), 215–222 (1976)
8. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40(6), 213–223 (2005)
9. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
10. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for C. SIGSOFT Softw. Eng. Notes 30(5), 263–272 (2005)
11. Tillmann, N., de Halleux, J.: Pex: White box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
12. Rosenblum, D.S., Weyuker, E.J.: Predicting the cost-effectiveness of regression testing strategies. SIGSOFT Softw. Eng. Notes 21(6), 118–126 (1996)
13. Z3: An efficient SMT solver (2010), http://research.microsoft.com/en-us/um/redmond/projects/z3/
14. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for Java. In: 20th International Conference on Automated Software Engineering, pp. 114–123. ACM, New York (2005)
15. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: 16th Annual Network & Distributed System Security Symposium, Internet Society (2008)
16. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–264. ACM, New York (2006)
17. Orso, A., Kennedy, B.: Selective capture and replay of program executions. SIGSOFT Softw. Eng. Notes 30(4), 1–7 (2005)

18. Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 380–403. Springer, Heidelberg (2006)
19. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. 17(9), 900–910 (1991)
20. Taneja, K., Xie, T.: DiffGen: Automated regression unit-test generation. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 407–410. IEEE Computer Society Press, Los Alamitos (2008)
21. Evans, R.B., Savoia, A.: Differential testing: A new approach to change detection. In: 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 549–552. ACM, New York (2007)
22. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 504–527. Springer, Heidelberg (2005)
23. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: 19th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, pp. 196–205. IEEE Computer Society, Los Alamitos (2004)
24. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: 18th ACM Symposium on Operating Systems Principles, pp. 57–72. ACM, New York (2001)
25. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 35–44. ACM, New York (2007)
26. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: MSeqGen: Object-oriented unit-test generation via mining source code. In: 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 193–202. ACM, New York (2009)

# Combining Static Analysis and Test Generation for C Program Debugging

Omar Chebaro[1,2], Nikolai Kosmatov[1], Alain Giorgetti[2,3], and Jacques Julliand[2]

[1] CEA, LIST, Software Safety Laboratory, PC 94, 91191 Gif-sur-Yvette France
`firstname.lastname@cea.fr`

[2] LIFC, University of Franche-Comté, 25030 Besançon Cedex France
`firstname.lastname@lifc.univ-fcomte.fr`

[3] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France

**Abstract.** This paper presents our ongoing work on a tool prototype called SANTE (Static ANalysis and TEsting), implementing a combination of static analysis and structural program testing for detection of run-time errors in C programs. First, a static analysis tool (Frama-C) is called to generate alarms when it cannot ensure the absence of run-time errors. Second, these alarms guide a structural test generation tool (PathCrawler) trying to confirm alarms by activating bugs on some test cases. Our experiments on real-life software show that this combination can outperform the use of each technique independently.

**Keywords:** all-paths test generation, static analysis, run-time errors, C program debugging, alarm-guided test generation.

## 1 Introduction

Software validation remains a crucial part in software development process. Software testing accounts for about 50% of the total cost of software development. Automated software validation is aimed at reducing this cost. The increasing demand has motivated much research on automated software validation. Two major techniques have improved in recent years, dynamic and static analysis. They arose from different communities and evolved along parallel but separate tracks. Traditionally, they were viewed as separate domains.

Static analysis examines program code and reasons over all possible behaviors that might arise at run time. Since program verification is in general undecidable, it is often necessary to use approximations. Static analysis is conservative and sound: the results may be weaker than desirable, but they are guaranteed to generalize to all executions. Dynamic analysis operates by executing a program and observing this execution. It is in general incomplete due to a big (or even infinite) number of possible test cases. Dynamic analysis is efficient and precise because no approximation or abstraction needs to be done: the analysis can examine the actual, exact run-time behavior of the program for the corresponding test case.

The pros and cons of the two techniques are apparent. If dynamic analysis detects an error then the error is real. However, it cannot in general prove the absence of errors. On the other hand, if static analysis reports a potential error, it may be a false alarm.

However, if it does not find any error (of a particular kind) in the overapproximation of program behaviors then the analyzed program clearly cannot contain such errors. Static and dynamic analysis have complementary strengths and weaknesses and can be both applied to program verification. Static analysis is typically used for proofs of correctness. Dynamic analysis demonstrates the presence of errors and increases confidence in a system.

Recently, there has been much interest in combining dynamic and static methods for program verification [1,2,3,4,5,6]. Static and dynamic analyses can enhance each other by providing valuable information that would otherwise be unavailable. This paper reports on an ongoing project that aims to provide a new combination of static analysis and structural testing of C programs. We implement our method using two existing tools: Frama-C, a framework for static analysis of C programs, and PathCrawler, a structural test generation tool.

Frama-C [7] is being developed in collaboration between CEA LIST and the ProVal project of INRIA Saclay. Its software architecture is plug-in-oriented and allows fine-grained collaboration of analysis techniques. Static analyzers are implemented as plug-ins and can collaborate with one another to examine a C program. Frama-C is distributed as open source with various plug-ins. Developed at CEA LIST, PathCrawler [8] is a test generation tool for C functions respecting *the all-paths criterion,* which requires to cover all feasible program paths, or *the $k$-path criterion,* which restricts the generation to the paths with at most $k$ consecutive iterations of each loop.

**Contributions.** This paper presents our ongoing work combining static analysis and structural test generation for validation of C programs, in particular, for detection of run-time errors. We call this technique *alarm-guided test generation*. Our ongoing implementation of this method, called SANTE, assembles two heterogeneous tools using quite different technologies (such as abstract interpretation and constraint logic programming). We evaluate our method by several experiments on real-life C programs, and compare the results with static analysis alone, test generation alone, and test generation guided by the exhaustive list of alarms for all potentially threatening statements. In all cases, our method outperforms the use of each technique independently.

The paper is organized as follows. Section 2 gives an overview of our method and its implementation in progress. Section 3 presents initial experiments illustrating the benefits of our approach. Section 4 briefly presents related work and concludes.

## 2 Overview of the Method

This section presents our method combining static analysis and test generation, and its ongoing implementation in a tool prototype SANTE[1] (Static ANalysis and TEsting) which uses Frama-C and PathCrawler tools. Our implementation choice was to connect PathCrawler and Frama-C via a new plug-in, and adapt PathCrawler to accept information provided by other plug-ins.

Algorithm 1 shows an overview of the method. SANTE takes as input the C program $P$ to be analyzed and the test context (denoted by *Context*) defining the function to

[1] The French word “santé” means “health”, and sometimes also “cheers!”.

**Algorithm 1.** Algorithm of the method

```
SANTE(P, Context)
1: $\Psi$ := VALUEANALYSIS(P, Context)
2: if $\forall i \in I, \Psi_i \equiv$ false then
3:     return proved                /* no alarms */
4: else
5:     $P'$ := ADDERRORBRANCHES($P, \Psi$)
6:     $B$ := PATHCRAWLER($P'$, Context)
7:     return $B$
8: end if
```

```
ADDERRORBRANCHES($P, \Psi$)
1: for all $i \in I$ do
2:     if $\Psi_i \equiv$ false then
3:         $\alpha'_i := \alpha_i$                /* no alarm for $\alpha_i$ */
4:     else
5:         $\alpha'_i$ := if($\Psi_i$) storeBugAndExit();else $\alpha_i$
6:     end if
7: end for
8: return $P' = \{\alpha'_i \mid i \in I\}$
```

be analyzed, domains of its input variables and preconditions. We denote by $\alpha_i, i \in I$ the statements of the program $P$. SANTE starts by analyzing the program with the value analysis plug-in of Frama-C. Based on abstract interpretation, this plug-in computes and stores supersets of possible value ranges of variables at each statement of the program. Among other applications, these over-approximated sets can be used to exclude the possibility of a run-time error. The value analysis is sound: it emits an alarm for an operation whenever it cannot guarantee the absence of run-time errors for this operation. It starts from an entry point in the analyzed program specified by the user, and unrolls function calls and loops. It memorizes abstract states at each statement and provides an interface for other plug-ins to extract these states.

The abstract states make it possible to extract $\Psi = \{\Psi_i \mid i \in I\}$, where $\Psi_i$ is the condition restricting the state before the statement $\alpha_i$ to an error state, in other words, describing the states leading to a possible run-time error at $\alpha_i$. For instance, for the statement `x=y/z;` the plug-in emits "*Alarm:* `z` *may be 0!*" and returns $\Psi_i \equiv (\mathtt{z} = 0)$ if 0 is contained in the superset of values computed for `z` before this statement. For the last statement in `int t[10]; ... t[n]=15;` the plug-in emits "*Alarm:* `t+n` *may be invalid!*" and returns $\Psi_i \equiv (\mathtt{n} < 0 \vee \mathtt{n} > 9)$ when it cannot exclude the risk of out-of-range index `n`. For `int* p; ... *(p+j)=10;` the plug-in emits "*Alarm:* `p+j` *may be invalid!*" if it cannot guarantee that `p+j` refers to a valid memory location. In the current version, the extraction of $\Psi_i$ is supported for division by 0 and out-of-range array index, and not yet fully supported for invalid pointers or non-initialized variables. If value analysis sees no risk of a run-time error at $\alpha_i$, then $\Psi_i \equiv$ *false*.

If all $\Psi_i \equiv$ *false*, i.e. no alarms were reported, then all possible program executions are error-free and the program is proved to contain no run-time errors. If some $\Psi_i$ is not trivial, we use the following technique called *alarm-guided test generation* (lines 5–6 in SANTE). We realize a specific instrumentation of $P$ represented here by the function ADDERRORBRANCHES. It takes as inputs the original program $P$ and the alarms $\Psi_i, i \in I$ for its statements, and returns a new program $P' = \{\alpha'_i \mid i \in I\}$. ADDERRORBRANCHES iterates over the statements $\alpha_i$ of $P$ and, if there is no alarm for $\alpha_i$, keeps $\alpha'_i = \alpha_i$. Otherwise it replaces the statement $\alpha_i$ by the statement

```
if( $\Psi_i$ ) storeBugAndExit(); else $\alpha_i$
```

In other words, if the alarm condition is verified, a run-time error can occur, so the function `storeBugAndExit()` reports a potential bug and stops the execution of the current test case. If there is no risk of run-time error, the execution continues normally and $P'$ behaves exactly as $P$.

Next, PathCrawler is called for $P'$. The PathCrawler test generation method [8] is similar to the so-called *concolic,* or *dynamic symbolic execution.* The user provides the C source code of the function under test. The generator explores program paths in a depth-first search using symbolic and concrete execution. The transformation of $P$ into $P'$ adds new branches for error and error-free states so that the PathCrawler test generation algorithm will automatically try to cover error states. It returns the list of detected bugs $B$ with error paths and inputs which confirms some alarms. Other alarms may remain unconfirmed due to various reasons: (1) this is a false alarm, (2) test generation timed/spaced out or (3) incomplete test selection strategy was used e.g. $k$-path.

## 3 Experiments

In this section, we compare our combined method with static analysis and with two test generation techniques used independently. The first testing technique is running PathCrawler with various strategies but without any information on threatening statements. The second one, denoted *all-threats*, runs PathCrawler in alarm-guided mode like SANTE, but for the exhaustive list of alarms for all potentially threatening statements (i.e. with potential risk of a run-time error).

We use five examples shown in Fig. 1 extracted from real-life software where bugs were previously detected. All bugs are out-of-range indices or invalid pointers. Examples 1–4 come from Verisec C analysis benchmark [9], example 5 from [10]. The columns of Fig. 1 respectively present the example number, its origin, the name of the analyzed function, the size of each example in lines of code, the total number of potential threats, the number of known bugs among them, and the results of value analysis. Fig. 2 compares SANTE to other test generation techniques. Its columns respectively show the example number, the PathCrawler strategy (test selection criterion) and the results for each method. The column 'safe' provides the number of threats proven unreachable by value analysis or by exhaustive all-paths testing when it terminates. The column 'unknown' provides the number of remaining unconfirmed alarms (relevant for value analysis, PathCrawler all-threats and SANTE). We also present, when relevant, the number of bugs detected, the number of treated paths, and full process duration. The strategy $k$-path is given for the minimal $k$ allowing to detect all bugs in SANTE. Experiments were conducted on an Intel Duo 1.66 GHz notebook with 1 GB of RAM with a 30 min timeout.

**SANTE vs. static analysis.** Fig. 1 shows that in most cases static analysis alone reduces the number of potential threats and proves that some of them are safe, but still generates many alarms. We see in Fig. 2 that SANTE confirms some alarms as real bugs, provides a test case activating each bug and leaves less unknown alarms.

| | origin | function name | size (loc) | all threats | known bugs | value analysis | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | safe | unknown | time |
| 1 | Apache | escape_absolute_uri (simplified) | 33 | 8 | 1 | 4 | **4** | 1s |
| 2 | Apache | escape_absolute_uri (full) | 97 | 16 | 1 | 11 | **5** | 1s |
| 3 | Spam Assassin | message_write | 55 | 17 | 2 | 2 | **15** | 1s |
| 4 | Apache | get_tag | 165 | 12 | 3 | 0 | **12** | 2s |
| 5 | QuickSort | partition | 50 | 8 | 1 | 4 | **4** | 1s |

**Fig. 1.** Examples and static analysis results

| | strategy | PathCrawler alone | | | PathCrawler all-threats | | | | | SANTE | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | bugs | paths | time | safe | unknown | bugs | paths | time | safe | unknown | bugs | paths | time |
| 1 | all-paths | **0** | 2164 | 14s | 7 | **0** | 1 | 3602 | **22s** | 7 | **0** | **1** | 2454 | 14s |
| | 3-path | **0** | 30 | ¡1s | 0 | **7** | 1 | 71 | **¡1s** | 4 | **3** | **1** | 45 | ¡1s |
| 2 | all-paths | **0** | 2023 | 10s | 15 | **0** | 1 | 3876 | **20s** | 15 | **0** | **1** | 2694 | 13s |
| | 10-path | **0** | 232 | 1s | 0 | **15** | 1 | 417 | **1s** | 11 | **4** | **1** | 325 | 1s |
| 3 | all-paths | **0** | 31917 | 311s | time / space out | | | | | 15 | **0** | **2** | 37967 | 523s |
| | 3-path | **0** | 12446 | 120s | 0 | **15** | 2 | 30977 | **558s** | 2 | **13** | **2** | 18874 | 215s |
| 4 | all-paths | time / space out | | | time / space out | | | | | time / space out | | | | |
| | 2-path | **1** | 26595 | 663s | 0 | **9** | 3 | 36690 | **870s** | 0 | **9** | **3** | 36690 | 872s |
| 5 | all-paths | **1** | 5986 | 33s | 7 | **0** | 1 | 15216 | **116s** | 7 | **0** | **1** | 11893 | 72s |
| | 2-path | **1** | 569 | 5s | 0 | **7** | 1 | 4509 | **25s** | 4 | **3** | **1** | 3319 | 18s |

**Fig. 2.** Experimental results for two test generation techniques and our combined method

**SANTE vs. PathCrawler alone.** SANTE detects more bugs than PathCrawler alone, and treats additional paths arising from error branches with reasonable extra time (Fig. 2, see for instance Ex. 3).

**SANTE vs. PathCrawler all-threats.** Alarm-guided test generation in SANTE only treats the alarms raised by value analysis while all-threats dully considers all potential threats. Thus test generation in SANTE considers less paths, detects the same number of bugs in less time and leaves less unknown alarms. It terminates in some cases where all-threats spaces/times out (Ex. 3). In the worst case, when static analysis can't filter any threat, SANTE can take as much time as all-threats (cf Ex. 4, 2-path).

Additional application of program slicing before alarm-guided test generation didn't show obvious gain here, because these examples were already simplified.

## 4 Related Work and Conclusion

**Closely related work.** Many static and dynamic analysis tools are well known and widely used in practice. Recently, several papers presented combinations of dynamic and static methods for program verification, e.g. [1,2,3,4,5,6]. Daikon [4] uses dynamic analysis to detect likely invariants. [5] compares two combined tools for Java: Check 'n'

Crash and DSD-Crasher. Our all-threats method is similar to [11], called *active property checking* in [6]. Synergy/Dash [3] and BLAST [2] combine testing and partition refinement for property checking. The idea of combining static analysis and testing for debugging was mentioned in [6] but was not implemented and evaluated.

**Conclusion.** We have presented our ongoing research on a new method combining static analysis and structural testing, as well as experimental results showing that this method is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than a concolic structural testing tool alone or guided by the exhaustive list of alarms for all potentially threatening statements. Static analysis alone will in general just generate alarms (some of which may be false alarms), whereas our method allows to confirm some alarms as real bugs and provides a test case activating each bug. This is done automatically, avoiding, at least for confirmed alarms, time-consuming alarm analysis by the validation engineer, requiring significant expertise, experience and deep knowledge of source code. Stand-alone test generation, when it is not guided by generated alarms for some statements, does not detect as many bugs as our combined method. When guided by the exhaustive list of alarms for all potentially threatening statements (not filtered by static analysis), test generation usually has to examine more infeasible paths and takes more time than our combined method (or even times/spaces out). In all cases, our method outperforms each technique used independently. Since complete all-paths testing is unrealistic for industrial software, it is also encouraging to see that realistic partial criteria (e.g. $k$-path) are very efficient in SANTE method. We expect that other testing techniques will also gain from the use of static analysis as concolic testing evaluated here.

Future work includes continuing research to eliminate unconfirmed alarms, to better support other alarm types (e.g. invalid pointers) and to integrate program slicing; experimenting with other coverage criteria (e.g. all-branches) and with breadth-first search; extending the SANTE implementation and comparing it with other tools and on more benchmarks.

**Acknowledgments.** The authors thank the members of the PathCrawler and Frama-C teams for providing the tools and support. Special thanks to Loic Correnson, Bernard Botella and Bruno Marre for their helpful advice and fruitful suggestions.

## References

1. Pasareanu, C., Pelanek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 52–66. Springer, Heidelberg (2005)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: Applications to software engineering. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
3. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: FSE, pp. 117–127 (2006)
4. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program 69(1-3), 35–45 (2007)

5. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 1–16. Springer, Heidelberg (2007)
6. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: EMSOFT, pp. 207–216 (2008)
7. Frama-C: Framework for static analysis of C programs (2007-2010), http://www.frama-c.com/
8. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST, pp. 70–78 (2009)
9. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE, pp. 389–392 (2007)
10. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 1–22. Springer, Heidelberg (2005)
11. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: CCS, pp. 322–335 (2006)

# Generating High-Quality Tests for Boolean Circuits by Treating Tests as Proof Encoding[⋆]

Eugene Goldberg and Panagiotis Manolios

Northeastern University, USA
{eigold,pete}@ccs.neu.edu

**Abstract.** We consider the problem of test generation for Boolean combinational circuits. We use a novel approach based on the idea of treating tests as a proof encoding rather than as a sample of the search space. In our approach, a set of tests is complete for a circuit $N$, and a property $p$, if it "encodes" a formal proof that $N$ satisfies $p$. For a combinational circuit of $k$ inputs, the cardinality of such a complete set of tests may be exponentially smaller than $2^k$. In particular, if there is a short resolution proof, then a small complete set of tests also exists. We show how to use the idea of treating tests as a proof encoding to directly generate high-quality tests. We do this by generating tests that encode mandatory fragments of *any* resolution proof. Preliminary experimental results show the promise of our approach.

## 1 Introduction

Although formal verification has made significant progress, simulation, due its scalability, is still the main workhorse of functional verification. An obvious drawback of simulation is that it only samples the search space and so may miss some bugs. Making simulation *complete* (i.e guaranteeing the lack of bugs) at the same time keeping the number of tests reasonably small is a very exciting goal.

Recent results [6] show that finding small complete test sets is actually possible. These results are based on the idea of treating a test set as an encoding of a formal proof (that the required property holds) rather than a sample of the search space. We will refer to this concept as Treating Tests as a Proof Encoding (TTPE). In particular, it was shown that to encode a resolution proof of $k$ resolutions one needs at most $2k$ tests. Importantly, a test set encoding a formal proof is *complete* (in the sense that no bug can be missed) and may be very small. Such a test set may be exponentially smaller than a trivial complete test set of $2^n$ tests (where $n$ is the number of inputs of $N$).

In this paper, we use TTPE for verification of combinational circuits. The generic problem here is to show that a single-output circuit $N$ always evaluates to 0 or to find a bug, an input assignment $\boldsymbol{x}$ such that $N(\boldsymbol{x})$=1. In principle, TTPE can be used for both proving that $N \equiv 0$ and for showing that $N$ is buggy by generating a sequence of tests until we either encode a proof or find a bug. In

⋆ This research was funded in part by NASA Cooperative Agreement NNX08AE37A.

more detail, suppose we generated tests $\boldsymbol{x_1}$, ..., $\boldsymbol{x_{k-1}}$, but they do not encode a proof and they have not found a bug. We generate a new test, $\boldsymbol{x_k}$, and check if $N(\boldsymbol{x_k}) = 1$. If so, $N$ is buggy. Otherwise, we check if the set $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}\}$ encodes a resolution proof. If it does, then $N \equiv 0$. Otherwise, we continue generating tests. This is a simplified overview of the procedure *ProofByTesting* described in Section 5.

An obvious question is: when does a set of tests encode a resolution proof? The key idea is to use tests to enable certain resolution steps. Thus, a set of tests encodes a resolution proof if the set of resolution steps allowed by the tests includes all the steps of the proof. A full account can be found in Section 4.

Unfortunately, no *efficient* procedure for checking if a set of tests encodes a resolution proof is currently known. However, we develop an efficient variation of *ProofByTesting* meant *only* for showing that $N$ is buggy. Generation of high-quality tests for checking if $N$ has a bug (*i.e.*, evaluates to 1 for some test) is *the problem we address in this paper*. Our approach is based on the idea of generating tests that encode "mandatory" fragments of a resolution proof. Given a CNF formula $F$, a particular class of complete assignments called boundary points of $F$ specify mandatory resolutions of a proof that $F$ is unsatisfiable [5]. In this paper, we show that boundary points can be used for generation of high quality tests.

The idea of extracting tests from boundary points is the first contribution of our paper. The second contribution is showing that TTPE can be used for building good tests indirectly, *i.e.*, without generating an explicit proof. Our third contribution is in giving experimental evidence that tests extracted from boundary points have high quality.

Given a CNF formula $F(v_1, \ldots, v_n)$, a $\mathit{lit}(v_i)$-boundary point $\boldsymbol{p}$ is an unsatisfying assignment such that all clauses of $F$ falsified by $\boldsymbol{p}$ share the same literal $\mathit{lit}(v_i)$ (where $\mathit{lit}(v_i)$ is $v_i$ or $\overline{v}_i$). Importantly, a $\mathit{lit}(v_i)$-boundary point $\boldsymbol{p}$ *mandates* a resolution on variable $v_i$ [5] (any proof that $F$ is unsatisfiable has to have a resolution on $v_i$ eliminating $\boldsymbol{p}$ as a $\mathit{lit}(v_i)$-boundary point). So one can use boundary points to encode mandatory fragments of a resolution proof that the design property specified by $F$ holds.

It is an open question whether a resolution proof can be encoded by boundary points alone (*i.e.*, whether resolutions mandated by boundary points always constitute a resolution proof). It was shown experimentally in [5] that for well structured proofs, the share of resolutions mandated by boundary points is high (90-100%). This implies that by generating boundary points one has a good chance to encode a proof or a large part of it.

Studying the relation of tests and proofs in propositional logic is important for at least three reasons. First, propositional logic plays an outstanding role in hardware verification. Second, it is also used in software verification. The state-of-the art SMT-solvers are based on propositional SAT-solvers. The latter are also extensively used in software verification systems like Alloy [10], CBMC [11], Java pathfinder [12]. Third, in many cases testing works well in verification of sequential circuits and programs. This implies that the TTPE approach may be applicable to logics more complex than propositional.

Since the method we introduce in this paper is heavily based on the TTPE approach, we describe the latter in Sections 2-5. We use a simple example to explain our definitions. Then in Section 6 we describe how tests are extracted from boundary points. In Section 7, we relate our approach with hardware testing based on the stuck-at fault model and with mutation-based testing. Finally, we give some experimental results and make conclusions.

## 2 Example

In this section, we introduce an example we will use extensively throughout the paper. Figure 1 shows a circuit called *miter* that is meant for equivalence checking of combinational circuits $M'$ and $M''$. In a miter, circuits $M'$ and $M''$ share the same set of inputs. Besides, the outputs of $M'$ and $M''$ feed an XOR gate (in our example, it is gate $G_6$). The circuits $M'$ and $M''$ are functionally inequivalent iff their miter evaluates to 1 (in this case there is an input assignment for which $M'$ and $M''$ produce different values and so the XOR gate evaluates to 1).

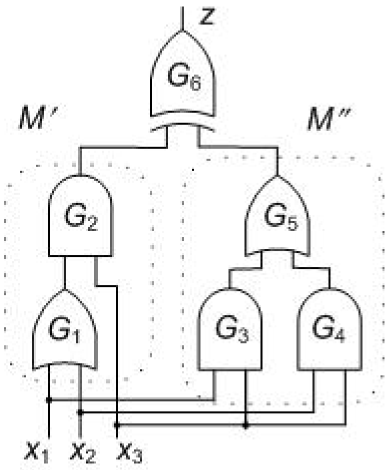


**Fig. 1.** Miter $N$ of functionally equivalent circuits $M'$ and $M''$

The miter $N$ shown in Figure 1, checks for equivalence circuits $M'$ and $M''$ implementing the expressions $(x_1 \vee x_2) \wedge x_3$ and $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ respectively. Since $M'$ and $M''$ are functionally equivalent, $N$ always evaluates to 0. The conventional wisdom is that to prove that $N$ implements constant 0 by simulation one has to generate $2^3 = 8$ tests. As we show later, for the circuit $N$ of Figure 1 there is a *complete* test set of only 5 tests. This test set is complete in the sense it encodes a resolution proof that a CNF formula $F$ specifying the query "Does $N(\boldsymbol{x})$ evaluate to 1 for some $\boldsymbol{x}$?" is unsatisfiable. These 5 tests are extracted from boundary points of $F$.

## 3 Some Basic Definitions

In this paper, we specify combinational circuits by CNF formulas. This section gives some relevant definitions.

**Definition 1.** *A* ***literal*** *lit($v_i$) of a Boolean variable $v_i$ is either $v_i$ itself (the* ***positive literal*** *of $v_i$) or the negation of $v_i$ denoted as $\overline{v}_i$ (the* ***negative literal*** *of $v_i$).*

**Definition 2.** *A* ***clause*** *is the disjunction of literals where no two (or more) literals of the same variable can appear. A clause consisting of only one literal is called* ***unit***. ***A CNF formula*** *is the conjunction of clauses. We will also view a CNF formula as* ***a set of clauses***.

**Definition 3.** *Given a CNF formula $F(v_1,\dots,v_n)$, a* ***complete assignment*** *(also called a* ***point****) is a mapping $\{v_1,\dots,v_n\} \rightarrow \{0,1\}$. Given a complete assignment $\boldsymbol{p}$ and a clause $C$, denote by $C(\boldsymbol{p})$ the value of $C$ when its variables are assigned as in $\boldsymbol{p}$. A clause $C$ is* ***satisfied*** *(respectively* ***falsified****) by a complete assignment $\boldsymbol{p}$, if $C(\boldsymbol{p}) = 1$ (respectively $C(\boldsymbol{p}) = 0$).*

**Definition 4.** *Given a CNF formula $F$, a* ***satisfying assignment p*** *is a complete assignment satisfying every clause of $F$. The* ***satisfiability problem (SAT)*** *is to find a satisfying assignment for $F$ or to prove that it does not exist.*

## 4 Encoding Resolution Proofs

In this section, we describe how a circuit is represented by a CNF formula and recall the basics of the resolution proof system. Then we describe how one can encode a resolution proof by complete assignments. (This is done as in [6] but without using the machinery of stable sets of points.) Finally, we define the notion of proof encoding in terms of tests.

### 4.1 CNF Representation of a Circuit

Typically, finding out if a property of a combinational circuit $M$ holds reduces to checking if a single-output circuit (derived from $M$) implements constant 0. (For instance, proving that combinational circuits $M$ and $M'$ are functionally equivalent comes down to checking if the miter $N$ of $M'$ and $M''$ always evaluates to 0, see Section 2).

Let $N$ be a single-output circuit of gates $G_1,\dots,G_m$ where the output of $G_m$ is also the output of $N$. Let the inputs of $N$, the outputs of gates of $G_1,\dots,G_{m-1}$ and the output of $G_m$ be denoted by $X = \{x_1,\dots,x_r\}$, $Y = \{y_1,\dots,y_{m-1}\}$ and $z$ respectively. Let $F_N$ be a CNF formula specifying $N$ that is obtained from $F$ using regular Tseitsin's transformations [9]. Namely, $F_N = F_{G_1} \wedge \dots \wedge F_{G_m}$ where $F_{G_i}$ is a CNF formula satisfied (respectively falsified) by the consistent (respectively the inconsistent) assignments to the pins of $G_i$.

*Example 1.* For the circuit $N$ of Figure 1, $X = \{x_1,x_2,x_3\}$, $Y = \{y_1,\dots,y_5\}$ (where $y_i$ specifies the output of gate $G_i, i = 1,\dots,5$) and variable $z$ specifies the output of $G_6$. The formula $F_N$ specifying $N$ can be represented as $F_{G_1} \wedge \dots \wedge F_{G_6}$. Formula $F_{G_1}$, for instance, specifies the OR gate $G_1$ and is equal to $(x_1 \vee x_2 \vee \overline{y}_1)$ $\wedge$ $(\overline{x}_1 \vee y_1)$ $\wedge$ $(\overline{x}_2 \vee y_1)$. For example, clause $(x_1 \vee x_2 \vee \overline{y}_1)$ is falsified by the inconsistent assignment $x_1 = 0, x_2 = 0, y_1 = 1$ and so is $F_{G_1}$.

The problem of finding an input assignment that sets the output of $N$ to 1 comes down to checking the satisfiability of the formula $F(X,Y,z) = F_N \wedge z$. A complete assignment $\boldsymbol{p}$ to the variables of $F$ can be represented as $(\boldsymbol{x},\boldsymbol{y},z^*)$ where $\boldsymbol{x},\boldsymbol{y},z^*$ are assignments to $X,Y$ and $z$ respectively. The part of $\boldsymbol{p}$ consisting of the assignments to the input variables (*i.e.*, the part $\boldsymbol{x}$) is called a *test*. We will denote it by $inp(\boldsymbol{p})$. If an assignment $\boldsymbol{p}$ satisfies all the clauses of $F_N$ (but may falsify the unit clause $z$ of $F$), the part $(\boldsymbol{y},z^*)$ of $\boldsymbol{p}$ is called the *correct execution trace* for the test $\boldsymbol{x}$=$inp(\boldsymbol{p})$.

In general, a complete assignment $\boldsymbol{p}$ falsifies clauses of $F_N$. In such a case, $(\boldsymbol{y},z^*)$ can be interpreted as a *faulty execution trace*. This means that at least one gate $G_i$ of $N$ produces the value that is different from the one implied by the input values of $G_i$.

*Example 2.* Let $\boldsymbol{p}$=$(x_1=0, x_2=1, x_3=1, y_1=1, y_2=1, y_3=0, y_4=1, y_5=1, z=0)$ a complete assignment to the variables of $F_N$ specifying circuit $N$ of Figure 1. The test $\boldsymbol{x}$ corresponding to $\boldsymbol{p}$ is $inp(\boldsymbol{p})$=$(x_1=0, x_2=1, x_3=1)$. Since $\boldsymbol{p}$ assigns consistent values to all 6 gates of $N$ it satisfies all the clauses of $F_N$ (the entire formula $F_N$ is given below in Example 3). So, in this case, $(y_1=1, y_2=1, y_3=0, y_4=1, y_5=1, z=0)$ is the correct execution trace for the test $\boldsymbol{x}$.

Now, let $\boldsymbol{p'}$=$(x_1=0, x_2=1, x_3=1, y_1=0, y_2=0, y_3=0, y_4=1, y_5=1, z=1)$. Point $\boldsymbol{p'}$ assigns consistent values to all the gates of $N$ but gate $G_1$ ($\boldsymbol{p'}$ falsifies the clause $(\overline{x}_2 \vee y_1)$ of $F_{G_1}$ given in Example 1). Point $\boldsymbol{p'}$ specifies the same test $\boldsymbol{x}$ as above and a faulty execution trace $(y_1=0, y_2=0, y_3=0, y_4=1, y_5=1, z=1)$.

## 4.2 Resolution Proofs

In this subsection, we recall the basics of the resolution proof system for propositional logic [2].

Resolution is a sound and complete proof system that has only one derivation rule called resolution.

**Definition 5.** *Let clauses $C'$,$C''$ have the opposite literals of variable $v_i$ (and no opposite literals of other variables). The* ***resolvent*** *$C$ of $C'$ and $C''$ on variable $v_i$ is the clause with all the literals of $C'$ and $C''$ but those of $v_i$. The clause $C$ is said to be obtained by a* ***resolution operation*** *on $v_i$. $C'$ and $C''$ are called the* ***parent clauses****.*

**Definition 6.** *([2]) Let $F$ be an unsatisfiable formula. Let $\{R_1, \ldots, R_k\}$ be a set of clauses such that*

- *each clause $R_i$ is obtained by a resolution operation where a parent clause is either a clause of $F$ or $R_j$, $j < i$;*
- *clauses $R_i$ are numbered in the derivation order;*
- *$R_k$ is an empty clause.*

*Then the set of $k$ resolutions that produced the resolvents $R_1, \ldots, R_k$ is called a* ***resolution proof*** *that $F$ is unsatisfiable.*

*Example 3.* Here we describe a resolution proof for the unsatisfiable CNF formula $F = F_N \vee z$ where $F_N = F_{G_1} \wedge \ldots \wedge F_{G_6}$ specifies the circuit $N$ of Figure 1. To describe subformulas $F_{G_i}$ one needs clauses $C_i, i = 1, \ldots, 19$ given below. Let $C_{20}$ denote the unit clause $z$. Then $F = C_1 \wedge \ldots \wedge C_{20}$.

$F_{G_1} = C_1 \wedge C_2 \wedge C_3$, $C_1 = x_1 \vee x_2 \vee \overline{y}_1$, $C_2 = \overline{x}_1 \vee y_1$,$C_3 = \overline{x}_2 \vee y_1$.
$F_{G_2} = C_4 \wedge C_5 \wedge C_6$, $C_4 = \overline{y}_1 \vee \overline{x}_3 \vee y_2$, $C_5 = y_1 \vee \overline{y}_2$, $C_6 = x_3 \vee \overline{y}_2$.
$F_{G_3} = C_7 \wedge C_8 \wedge C_9$, $C_7 = \overline{x}_1 \vee \overline{x}_3 \vee y_3$, $C_8 = x_1 \vee \overline{y}_3$, $C_9 = x_3 \vee \overline{y}_3$.
$F_{G_4} = C_{10} \wedge C_{11} \wedge C_{12}$, $C_{10} = \overline{x}_2 \vee \overline{x}_3 \vee y_4$, $C_{11} = x_2 \vee \overline{y}_4$, $C_{12} = x_3 \vee \overline{y}_4$.
$F_{G_5} = C_{13} \wedge C_{14} \wedge C_{15}$, $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$, $C_{14} = \overline{y}_3 \vee y_5$,$C_{15} = \overline{y}_4 \vee y_5$.
$F_{G_6} = C_{16} \wedge C_{17} \wedge C_{18} \wedge C_{19}$, $C_{16} = y_2 \vee \overline{y}_5 \vee z$, $C_{17} = \overline{y}_2 \vee y_5 \vee z$,
$C_{18} = y_2 \vee y_5 \vee \overline{z}$, $C_{19} = \overline{y}_2 \vee \overline{y}_5 \vee \overline{z}$.

A resolution proof $R$ that $F$ is unsatisfiable is given in Figure 2 as a DAG whose nodes are shown as ovals. Each non-leaf node corresponds to a resolution operation over the parent clauses specified by the preceding nodes. The proof $R$ is obtained by a version of a SAT-solver with conflict driven clause learning [7]. $R$ is partitioned into three chains of resolutions corresponding to the three conflicts (backtracks) that occurred when solving the formula $F$. Each chain describes derivation of a conflict clause shown in a dotted oval.

Empty ovals correspond to resolvents that are used only once right after they are generated. All the resolvents of the proof $R$ can be easily reproduced by performing the sequence operations specified by the graph of Figure 2. For instance, the first empty oval of the leftmost chain corresponds to resolving clause $C_8 = x_1 \vee \overline{y}_3$ with $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$ (on variable $y_3$) and producing the resolvent $x_1 \vee y_4 \vee \overline{y}_5$. The latter is then resolved with the clause $C_{11} = x_2 \vee \overline{y}_4$ (on variable $y_4$) producing the resolvent $x_1 \vee x_2 \vee \overline{y}_5$ corresponding to the next empty oval and so on. Eventually, the conflict clause $C_{21} = y_1 \vee \overline{z}$ is derived. In the middle resolution chain, the conflict clause $C_{22} = x_3 \vee \overline{z}$ is generated. Finally, the empty conflict clause $C_{23}$ is derived in the rightmost resolution chain.

## 4.3 Proof Encodings in Terms of Points

In this subsection, we explain what it means for a set of points to encode a resolution proof.

**Definition 7.** *Let $C',C''$ be two clauses and $\boldsymbol{p'}$ and $\boldsymbol{p''}$ be two complete assignments such that*

- *$\boldsymbol{p'}$ and $\boldsymbol{p''}$ are only different in the value of a variable $v_i$*
- *$C'(\boldsymbol{p'}) = C''(\boldsymbol{p''})=0$ and $C'(\boldsymbol{p''}) = C''(\boldsymbol{p'})=1$*

*Then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are said to **legalize** the resolution of $\boldsymbol{C'}$,$\boldsymbol{C''}$ on $v_i$.*

The two conditions of Definition 7 imply that $C'$,$C''$ have opposite literals of exactly one variable (which is $v_i$). This means that clauses $C'$,$C''$ indeed can be resolved on $v_i$. Intuitively, the existence of points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ satisfying the conditions of Definition 7 means that a clause (the resolvent of $C'$,$C''$) is implied by $C' \wedge C''$. In the approach based on the notion of a stable set of points [6], this implication is a theorem (that can be easily proved).

*Example 4.* Let us consider the first resolution operation of the proof $R$ described in Example 3 (*i.e.*, resolution over $C_8 = x_1 \vee \overline{y}_3$ and $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$). Let point

$\boldsymbol{p'}$ be $(x_1 = 0, .., y_3 = 1, y_4 = 0, y_5 = 1, ..)$ (the values of the missing variables can be chosen arbitrarily). Let $\boldsymbol{p''}$ be obtained from $\boldsymbol{p'}$ by flipping the value of $y_3$. Then $C_8(\boldsymbol{p'})$=$C_{13}(\boldsymbol{p''})$=0 and $C_8(\boldsymbol{p''})$=$C_{13}(\boldsymbol{p'})$=1. So $\boldsymbol{p'}$ and $\boldsymbol{p''}$ legalize the resolution over $C_8$ and $C_{13}$.

**Definition 8.** *Let $F$ be an unsatisfiable CNF formula and $P$ be a set of points. Let $R = \{R_1, \ldots, R_k\}$ be a resolution proof. We will say that $P$* ***encodes proof*** *$R$ if each of $k$ resolutions of $R$ is legalized by some points $\mathbf{p'}$ and $\mathbf{p''}$ of $P$.*

Informally, the fact that $P$ encodes a resolution proof means that the former is large enough to legalize resolutions comprising a proof that $F$ is unsatisfiable. Definitions 7 and 8 imply that to encode a resolution proof $R = \{R_1, \ldots, R_k\}$ one needs a set $P$ of at most $2k$ points (two points for each resolution). In reality, this number may be smaller because the same point of $P$ may participate in legalization of more than one resolution operation.

**Definition 9.** *Let $N$ be a single-output circuit and $T$ be a set of tests $\{\boldsymbol{x_1},\ldots,\boldsymbol{x_s}\}$. We will say that $T$ encodes a resolution proof $R$ that $F_N \wedge z$ is unsatisfiable if there is a set of points $P = \{\boldsymbol{p_1},\ldots,\boldsymbol{p_m}\}$ such that $P$ encodes $R$ and each test $\boldsymbol{x_i}$ of $T$ is the input part of a point $\boldsymbol{p_j}$ of $P$.*

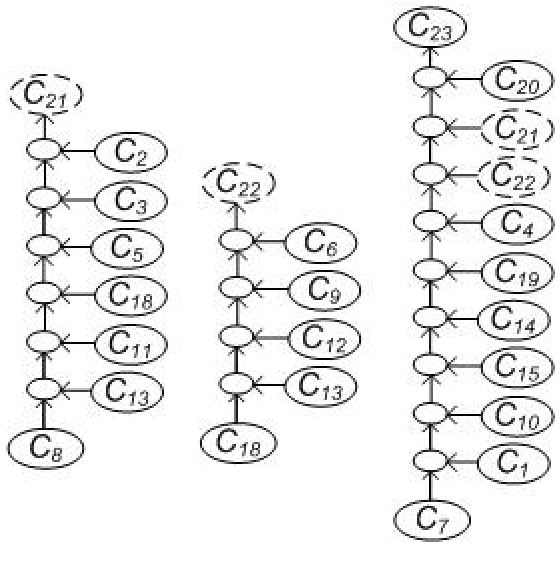


**Fig. 2.** A resolution proof that $F = F_N \wedge z$ is unsatisfiable

Definitions 8 and 9 imply that there is always a test set encoding a resolution proof $R$ that is at most two times the size of $R$.

*Example 5.* The proof $R$ of Figure 2 consists of 19 resolutions. So there is a set of at most 38 points encoding it. (In this small example, the size of the proof is larger than 8, which is the total number of assignments to 3 inputs. However, it is not unusual to have a proof, say, of $10^6$ resolutions for a circuit with 1000 inputs.) Since different points $\boldsymbol{p'}$ $\boldsymbol{p''}$ may have the same input part one may need less than 8 tests to encode $R$.

For instance, it can be shown that the following set of 5 tests $\boldsymbol{x_1}$=$(x_1 = 0, x_2 = 1, x_3 = 0)$, $\boldsymbol{x_2}$=$(x_1 = 0, x_2 = 0, x_3 = 1)$, $\boldsymbol{x_3}$=$(x_1 = 1, x_2 = 1, x_3 = 0)$, $\boldsymbol{x_4}$=$(x_1 = 1, x_2 = 0, x_3 = 1)$, $\boldsymbol{x_5}$=$(x_1 = 0, x_2 = 1, x_3 = 1)$ encodes the proof $R$. That there is a set of points $P$ legalizing the 19 resolutions of $R$ such that the number of different input parts of points from $P$ is 5.

## 5 Test Generation by Proof Encoding

In this section, we describe a procedure (called *ProofByTesting*) that, given a single-output combinational circuit $N$ checks if $N \equiv 0$. Our objective here is threefold. First, we want to illustrate the point that the idea of TTPE is applicable to both buggy and correct circuits. ($N$ is buggy when $N(\boldsymbol{x})=1$ for some input assignment $\boldsymbol{x}$.) Second, even though *ProofByTesting* is not practical, it illustrates the point that one can have a complete test set that is smaller than $2^n$ (where $n$ is the number of inputs of $N$). Third, the TCBP procedure described in Subsection 6.3 is a variation of *ProofByTesting* made efficient by removing checks that a set of points encodes a proof. (So this variation is limited to finding bugs in $N$).

```
c the procedure checks if $N \equiv 0$
$ProofByTesting(N)$
{ $P = \emptyset$, $F = F_N \wedge z$;
while(true)
  {$\boldsymbol{p}$ = $gen\_pnt(F)$;
  $\boldsymbol{x}$ = $inp(\boldsymbol{p})$;
  if ($N(\boldsymbol{x})$ == 1) return($no$);
  $P = P \cup \{\boldsymbol{p}\}$;
  while (true)
    {($C$,$Exst$)= $new\_legal\_res(P,F)$;
    if ($Exst$==$false$) break;
    if ($C$ == $empty$) return($yes$);
    $F = F \cup \{C\}$; }}}
```

**Fig. 3.** A procedure for generating tests in the process of encoding a proof

Pseudocode of the *ProofByTesting* procedure is shown in Figure 3. First, it builds CNF formula $F$ as described in Subsection 4.1. In the outer loop, *ProofByTesting* generates a point $\boldsymbol{p}$ and extracts its input part $\boldsymbol{x}$. (We assume here that the same point is not generated more than once). If $\boldsymbol{x}$ is a counterexample, the procedure returns *no*. Otherwise, $\boldsymbol{p}$ is added to the set of points $P$ and the inner loop begins. In this loop *ProofByTesting* checks if $P$ encodes a resolution proof.

First, *ProofByTesting* arbitrarily picks a new resolvent $C$ obtained by a resolution legalized by points of $P$. (We assume here that no resolvent is generated if it is implied by an existing clause of $F$.) If all legal resolvents have been generated, *ProofByTesting* leaves the inner loop to generate a new point $\boldsymbol{p}$. Otherwise, it checks if $C$ is an empty clause. If it is, the answer *yes* is returned. (The set of resolvents added to $F$ contains a resolution proof and so $N \equiv 0$. This proof is encoded by points of $P$. The input parts of points from $P$ form a test set encoding a resolution proof.) If $C$ is not empty, it is added to $F$ and a new iteration of the inner loop begins.

## 6 Extracting Tests from Boundary Points

In this section, we recall the definition and some properties of boundary points and describe the idea of using such points for proof encoding. Then we give a simple algorithm (called *TCBP*) for generation of tests extracted from boundary points. *TCBP* is a variation of *ProofByTesting* described in the previous section. Instead of running inefficient checks if a set of points encodes a resolution proof, *TCBP* generates points that encode mandatory parts of a resolution proof. (Since *TCBP* does not encode a complete proof, it can be used only for finding bugs.)

### 6.1 Definition of Boundary Points and Some Useful Properties

**Definition 10.** *Denote by Unsat($\boldsymbol{p}$,$F$) the set of clauses of a CNF formula $F$ falsified by a complete assignment $\boldsymbol{p}$.*

**Definition 11.** *Given a CNF formula $F$, a complete assignment $\boldsymbol{p}$ is called a lit($v_i$)-boundary point, if Unsat($\boldsymbol{p}$,$F$) $\neq \emptyset$ and every clause of Unsat($\boldsymbol{p}$,$F$) contains literal lit($v_i$).*

*Example 6.* The point $\boldsymbol{p}=(x_1=0, x_2=1, x_3=0, y_1=1, y_2=0, y_3=1, y_4=0, y_5=1, z=1)$ falsifies only the clauses $C_8 = x_1 \vee \overline{y}_3$ and $C_9 = x_3 \vee \overline{y}_3$ of the formula $F$ of Example 3. These two clauses share literal $\overline{y}_3$. So $\boldsymbol{p}$ is a $\overline{y}_3$-boundary point.

**Definition 12.** *Denote by Bnd_pnts($F$) the set of all boundary points of $F$.*

**Definition 13.** *Let $\boldsymbol{p}$ be a complete assignment. Denote by flip($\boldsymbol{p}$,$v_i$) the point obtained from $\boldsymbol{p}$ by flipping the value of $v_i$.*

The proposition below explains why studying boundary points is important.

**Proposition 1.** *([5]) If Bnd_pnts($F$) $= \emptyset$ , then $F$ is unsatisfiable.*

Proposition 1 implies that for a satisfiable formula $F$, *Bnd_pnts*$(F) \neq \emptyset$. In particular, it is not hard to show [5] that if $F(\boldsymbol{p'})$=0, $F(\boldsymbol{p''})$=1 and $\boldsymbol{p''}$= *flip*($\boldsymbol{p'}$,$v_i$), then $\boldsymbol{p'}$ is a *lit*($v_i$)-boundary point. (This explains the name "boundary point".) Another interesting fact is that if $\boldsymbol{p'}$ is a $v_i$-boundary point, the point $\boldsymbol{p''}$= *flip*($\boldsymbol{p'}$,$v_i$) is either a satisfying assignment or a $\overline{v}_i$-boundary point [5]. So for an unsatisfiable formula all boundary points come in pairs. We will refer to $\boldsymbol{p'}$ and $\boldsymbol{p''}$ as *symmetric* $v_i$-boundary and $\overline{v}_i$-boundary points.

Let $\boldsymbol{x}$ be the test corresponding to a *lit*($v_i$)-boundary point $\boldsymbol{p}$ (*i.e.*, $\boldsymbol{x}$ = *inp*($\boldsymbol{p}$)) where $v_i$ is not variable $z$. Then the part ($\boldsymbol{y}$,z) of $\boldsymbol{p}$ specifies a faulty execution trace for test $\boldsymbol{x}$. Namely, at least one gate of $N$ whose output/input variable is specified by $v_i$ produces the wrong output value (which is the negation of the value implied by the input values of this gate).

*Example 7.* Consider the $\overline{y}_3$-boundary point $\boldsymbol{p}=(x_1=0, x_2=1, x_3=0, y_1=1, y_2=0, y_3=1, y_4=0, y_5=1, z=1)$ of Example 6. It specifies test $\boldsymbol{x}=(x_1=0, x_2=1, x_3=0)$ and the execution trace $(y_1 = 1, y_2 = 0, y_3 = 1, y_4 = 0, y_5 = 1, z = 1)$. In this trace, the AND gate $G_3$ (whose output is described by $y_3$) produces the wrong output value 1 for the input values $x_1 = 0$, $x_3 = 0$ . (All the other gates produce output values implied by the input values of these gates.)

### 6.2 Encoding Resolutions Proofs by Boundary Points

Let $\boldsymbol{p'}$ and $\boldsymbol{p''}$ be symmetric $v_i$-boundary and $\overline{v}_i$-boundary points of $F$. It is not hard to show [5] that any clause $C'$ falsified by $\boldsymbol{p'}$ can be resolved on variable $v_i$ with any clause $C''$ falsified by $\boldsymbol{p''}$. This resolution produces a clause that is falsified by both $\boldsymbol{p'}$ and $\boldsymbol{p''}$ and does not have variable $v_i$. Then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are not *lit*($v_i$)- boundary points of $F \wedge C$. Adding a clause to $F$ can only eliminate some boundary points (but cannot produce new ones). So $Bnd\_pnts(F \wedge C) \subset Bnd\_pnts(F)$. We will refer to this process of removing boundary points by adding clauses implied by $F$ as *boundary point elimination*. (Note that adding $C$ to $F$ may also eliminate $lit(v_i)$-boundary points different from $\boldsymbol{p'}$ and $\boldsymbol{p''}$.)

Let $R_1, \ldots, R_k$ be a resolution proof where $R_k$ is an empty clause. Note that $Bnd\_pnts(F \wedge R_k) = \emptyset$. (By definition, if $p$ is a $lit(v_i)$-boundary point, every clause falsified by $\boldsymbol{p}$ has to have at least one literal, *i.e.*, $lit(v_i)$.) This means that every $lit(v_i)$-boundary point $\boldsymbol{p}$ of the initial formula $F$ is eventually eliminated. Then there is a resolvent $R_{m+1}$ such that $\boldsymbol{p}$ is a $lit(v_i)$-boundary point for $F \wedge R_1 \wedge \ldots \wedge R_m$ but not for $F \wedge R_1 \wedge \ldots \wedge R_{m+1}$. It was shown in [5] that a $lit(v_i)$-boundary point $\boldsymbol{p}$ is eliminated in the proof only by a resolution on variable $v_i$. In other words, a $lit(v_i)$-boundary point *mandates* a resolution on $v_i$. This fact is the foundation for using boundary points to encode resolution proofs.

If $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are symmetric $v_i$-boundary and $\overline{v}_i$-boundary points and they are eliminated by adding to $F$ the resolvent of $C'$ and $C''$ on $v_i$, then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ *legalize* this resolution (because $\boldsymbol{p'}$ , $\boldsymbol{p''}$ and $C'$ and $C''$ satisfy both conditions of Definition 7). If $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are symmetric boundary points legalizing a resolution on $v_i$, *every proof* has to contain a resolution on $v_i$ (but not necessarily the resolution of $C'$ and $C''$) In the general case, *i.e.*, when $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are not symmetric boundary points, they may legalize a resolution on variable $v_i$ even if there are proofs that have no resolutions on $v_i$. So proof encodings by boundary points are much closer related to proofs than encodings by arbitrary points.

It is not clear yet if one can encode an entire resolution proof using only boundary points. (It may be the case that some resolution operations of a proof can be legalized only by non-boundary points). However, the experimental study of [5] showed that for well structured proofs the ratio of resolutions that could be legalized by boundary points was close to 100%. This implies that (at least for the formulas of [5]) using boundary points one can encode an entire proof or a large part thereof.

*Example 8.* Every resolution operation of the proof $R$ described in Example 3 eliminates a boundary point (that has not been eliminated by previous resolutions). The set of 5 tests given in Example 5 was actually built by a program that extracted tests from boundary points eliminated by resolutions of $R$. For example, the $\overline{y}_3$-boundary point $\boldsymbol{p'}=(x_1 = 0, x_2 = 1, x_3 = 0, y_1 = 1, y_2 = 0, y_3 = 1, y_4 = 0, y_5 = 1, z = 1)$ introduced in Example 6 and the symmetric $y_3$-boundary point $\boldsymbol{p''}=flip(\boldsymbol{p'},y_3)$ are eliminated by the first resolution of $R$. (The latter resolves clauses $C_8 = x_1 \vee \overline{y}_3$ and $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$ on variable $y_3$.) Points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ legalize this resolution.

## 6.3 Extraction of Tests from Boundary Points

The procedure for extraction of tests from boundary points called *TCBP* (Testing based on Computation of Boundary Points) is shown in Figure 4. Given a single-output circuit $N$, specified by a CNF formula $F_N$, the *TCBP* procedure generates a set of tests to check if $N$ evaluates to 1. (This comes down to checking the satisfiability of CNF formula $F = F_N \wedge z$.) *TCBP* terminates if a test is found for which $N$ evaluates to 1 or if the number of generated tests exceeded a threshold. *TCBP* records the set of all generated tests. (The reason is as follows. Our experiments showed that due the high quality of tests generated by *TCBP*, even if a test is unsuccessful for $N$ it may detect a bug in a modified version $N$. This modification may correspond, for example, to a wrong design change.)

```
c $F = F_N \wedge z$
$TCBP(F,T)$
{ $count$=0; $T = \emptyset$;
while ($count < thresh$)
  {$v_i$= $pick\_var(F)$;
  ($ans$,$\boldsymbol{p}$) = $BndPnt(F,v_i,lim)$;
  if ($ans < success$) continue;
  else $count$++;
  $\boldsymbol{x}$=$inp(\boldsymbol{p})$; $T = T \cup \{\boldsymbol{x}\}$;
  if ($simulate(\boldsymbol{x},F)$ == $yes$)
    return ($found$);
  $eliminate(\boldsymbol{p},F)$;}
return($not\_found$);}
```

**Fig. 4.** *TCBP* procedure

The main work is done in the 'while' loop. First a variable $v_i$ of $F$ is picked randomly. Then the procedure *BndPnt* is called to find a $lit(v_i)$-boundary point. *lim.* If no boundary point is found by *BndPnts* within time limit *lim*, a new iteration of the 'while' loop is started. (In our experiments, *lim* was set to 10 sec.) Otherwise, a test $\boldsymbol{x}$ is constructed as the input part of the boundary point $\boldsymbol{p}$ found by *BndPnt.* This test is used for simulation. In terms of SAT, simulation comes down to adding the set $U$ of unit clauses encoding test $\boldsymbol{x}$ (*i.e.*, satisfied by the assignments of $\boldsymbol{x}$) to the formula $F$ and running Boolean Constraint Propagation (BCP).

If the circuit $N$ is deterministic, then BCP over the formula $F = F_N \wedge U \wedge z$ results in assigning a value to the output variable $z$. If $z$=1 (respectively $z$= 0), the test $\boldsymbol{x}$ is a counterexample (respectively not a counterexample). In experiments, we used faults that may make the behavior of a gate of $N$ non-deterministic. Then, variable $z$ may not get assigned after BCP is over. If this is the case, a SAT-solver was used to finish the instance (*i.e.*, to prove that $F$ was unsatisfiable if its input variables were assigned according to $\boldsymbol{x}$ or to find a satisfying assignment). If such an assignment was found, this meant that $\boldsymbol{x}$ detected the fault that the gate with non-deterministic behavior produced the wrong output value (*i.e.*, produced the negation of the value implied by the input assignments for the gate without the fault).

```
$BndPnt(F,v_i,lim)$
{$G = F \setminus (F_{v_i} \cup F_{\overline{v}_i})$.
($ans$,$\boldsymbol{p}$)=$sat(G,lim)$;
return(($ans$,$\boldsymbol{p}$));}
```

**Fig. 5.** *BndPnt* procedure

If $\boldsymbol{x}$ fails to detect a fault, the $lit(v_i)$-boundary point $\boldsymbol{p}$ from which $\boldsymbol{x}$ was extracted is eliminated by adding a resolvent on variable $v_i$. (No particular heuristic was used to pick the pair of clauses to be resolved). The test $\boldsymbol{x}$ is added to the set of tests $T$ and a new iteration of the 'while' loop starts.

The procedure for generation of a $lit(v_i)$-boundary point (called *BndPnt*) is given in Figure 5. First, the CNF formula $G$ is obtained from $F$ by removing the clauses having literal $v_i$ (denoted by $F_{v_i}$) and $\overline{v}_i$ (denoted by $F_{\overline{v}_i}$). Then a SAT-solver *sat* is called to check if $G$ is satisfiable. If a satisfying assignment $\boldsymbol{p}$ is found, it means that one of the following three possibilities occurred: a) $F_{v_i}(\boldsymbol{p}) = 0$ and hence $\boldsymbol{p}$ is a $v_i$-boundary point (because it falsifies only clauses of $F$ that have literal $v_i$); b) $F_{\overline{v}_i}(\boldsymbol{p})$=0 and $\boldsymbol{p}$ is a $\overline{v}_i$-boundary point; c) $F(\boldsymbol{p})$=1 and $\boldsymbol{p}$ is a satisfying assignment (never happened in our experiments).

If the SAT-solver *sat* fails to find a boundary point it may mean that the run time exceeded *lim* or that formula $G$ (and hence $F$) is unsatisfiable. (The latter never happened in our experiments because we considered only satisfiable formulas $F$ there).

## 7 Some Background

Methods of combining test generation and formal methods have been studied in many papers (e.g. [4,8] to name a few). In this section, for the lack of space, we only mention the work directly related to generation of tests extracted from boundary points (namely, generation of tests detecting hardware faults [1] and software mutations [3]).

Identification of defective chips is usually done by running tests that detect faults of a particular fault model [1]. In many cases, such a fault model may have little to do with what really happens on a defective chip. It is just used because tests detecting faults of this model are good at finding real faults. The most popular fault model of that kind is the stuck-at model. It describes the situation when a line of a gate is stuck at a constant value 0 or 1.

There is a tight relation between tests detecting stuck-at faults in a circuit $M'$ and boundary points of a CNF formula $F = F_N \wedge z$. Here $N$ is the miter of circuits $M'$ and $M''$ (like the one shown in Figure 1) where $M''$ is a copy of circuit $M'$. Variable $z$ specifies the output of $N$.

Consider, for example, a stuck-at-0 fault $\phi$ on the output line of AND gate $G'$ of $M'$. Let $y_i$,$y_j$ be the input variables of $G'$ and $y_k$ be its output variable. Denote by $M'_\phi$ the circuit $M'$ with fault $\phi$. A test $\boldsymbol{x_\phi}$ detecting $\phi$ (*i.e.*, detecting that $y_k \equiv 0$) has to assign $y_i$ and $y_j$ to 1. Then the gate $G'$ of $M'_\phi$ and its counterpart $G''$ in $M''$ produce different output values (0 and 1 respectively).

Let $\boldsymbol{p}$=($\boldsymbol{x_\phi}$,$\boldsymbol{y}$,1) be the point where ($\boldsymbol{y}$,1) is the correct execution trace for test $\boldsymbol{x_\phi}$ for the miter of $M'_\phi$ and $M''$. Then $\boldsymbol{p}$ is an $y_k$-boundary point for the formula $F$. Indeed, since $y_i = 1$,$y_j = 1$,$y_k = 0$ in $\boldsymbol{p}$, the latter falsifies clause $C = \overline{y}_i \vee \overline{y}_j \vee y_k$ of the formula $F_{G'}$ specifying gate $G'$. Since ($\boldsymbol{y}$,1) is the *correct* execution trace for the miter of $M'_\phi$ and $M''$, all the other gates of the miter $N$ of $M'$ and $M''$ are assigned correctly. It means that $\boldsymbol{p}$ satisfies all the clauses of the formula $F$ but $C$.

Summarizing, one can view introduction of stuck-at faults as a way to generate boundary points of formula $F$ (describing the miter of two *correct* copies of the same circuit). Importantly, the stuck-at fault model has been successfully used in industry for three decades, which serves as an indirect evidence of the quality of tests extracted from boundary points (at least for hardware testing).

The method of introducing mutations into a program has a lot of similarity with identification of defective chips based on fault models. In particular, the mutation operator replacing a logical subexpression with constant 'true' or 'false' introduces a “stuck-at fault” into a program. Another interesting similarity is that fault injection into a circuit (respectively introduction of a mutation into a program) can be used to identify redundancy in the circuit (respectively in the program). It is not hard to show that the absence of $lit(v_i)$- boundary points in a CNF formula $F$ means that the clauses of $F$ with variable $v_i$ can be removed from $F$ without changing its satisfiability. One can conjecture that similarly to hardware manufacturing testing, using mutations is just a way to produce tests that could have been extracted from some sort of boundary points of a formula describing the “correct” program (*i.e.*, the one without a mutation).

## 8 Experimental Results

In this section, we give some preliminary experimental results. Our intention here is just to check the idea of using tests extracted from boundary points by considering a simple example that mimics a hard real-life problem. This problem is verification of arithmetic devices embedded into control logic. (The famous Pentium bug was caused by failing to solve an instance of this problem.) In the experiments, we tested the miter $N$ of a faulty and correct circuits denoted by $M_f$ and $M$ respectively (an example of a miter is shown in Figure 1). The circuit $M$ contained a large arithmetic component. The circuit $M_f$ was a copy of $M$ with a fault introduced into the arithmetic component.

In the experiments we wanted to demonstrate the following two properties of tests built by TCBP (*i.e.*, extracted from boundary points). First, even though generation of such tests takes time their quality is much higher than that of random tests. Second, even unsuccessful TCBP tests meant for verification of $N$ (*i.e.*, tests for which $N$ evaluates to 0) have a high chance to find a bug in a modified version of $N$ if this modification is small. (In particular, we extracted tests from boundary points of an *unsatisfiable* formula $F$ describing the miter $N$ of two *identical* copies of circuit $M$. These tests were very effective in finding bugs when one copy of $M$ was replaced in $N$ with a faulty circuit $M_f$. We do not report this part of experiment for the lack of space.) From a practical point of view this means that the cost of tests generated by TCBP can be amortized over many design steps (due to reusing tests generated at previous design steps).

As a fault, we considered adding a literal to a clause of the CNF formula $F_M$ specifying circuit $M$. (We found the bugs of this kind to be very easy to introduce and very hard to detect.) The resulting formula $F_{M_f}$ specified the faulty circuit $M_f$. Adding a literal to a clause describing a gate, makes the latter behave non-deterministically. (Consider the clause $C = \overline{y}_i \vee \overline{y}_j \vee y_k$ of $F_M$ requiring that when inputs $y_i$ and $y_j$ of an AND gate $G_k$ are set to 1, the output of $G_k$ specified by $y_k$ is 1 too. After adding literal $\overline{y}_m$ to $C$, where $y_m$ specifies the output of gate $G_m$, the output of $G_k$ can take an arbitrary value, when $y_i = y_j = 1$ and $y_m = 0$.) A test $\boldsymbol{x}$ is considered to have detected a fault in gate $G_k$ of $M_f$, if the miter of $M_f$ and $M$ evaluates to 1 when $G_k$ produces the *wrong* value, *i.e.*, the negation of the value implied by the values of its inputs.

We compared *TCBP* with two extremes of functional verification: random testing (an instance of pure simulation) and checking the satisfiability of the miter $N$ by a SAT-solver (an instance of pure formal verification). For testing the satisfiability of $F$ in one SAT check we used SAT-solvers *Satzilla*, *Precosat* and *Mxc* that won the first, second and third places in the SAT-2009 competition [13] in the industrial category for satisfiable formulas. We also used *Precosat* in *TCBP* for finding boundary points.

We ran two experiments that were meant to probe the regions that are good and bad for random testing. In the first experiment, the circuit $M$ consisted only of an arithmetic component (good for random testing). In the second experiment, $M$ consisted of an arithmetic component feeding an input of a multi-input AND gate (bad for random testing).

The results of the first experiment are described in Table 1. In this experiment, circuit $M$ implemented the functionality of a medium bit of a 128-bit multiplier. (The size of the CNF formula $F$ specifying the miter of $M$ and $M_f$ was about 114,000 variables and 437,000 clauses. The formulas of Table 2 had about the same size.) We cherry picked 5 faults that were easy for random testing and hard for SAT-solving. The runtimes of SAT-solvers are shown in columns 2,3,4. (The timeout was set to 1 hour.) When using random testing, for every fault we generated 10 sets of random tests, the two columns of "random_testing" showing the average run time and average test set size.

The last three columns of Table 1 show the results of *TCBP*. To mitigate the influence of chance in picking boundary points, *TCBP* was given 10 tries to find tests for the set of 5 faults. Before generating tests for fault number $i$ ($i$=2,..,5), the tests generated for faults 1,..$i$-1 (*in the same try*) were applied. If an old test detected the fault no new tests were generated. In such a case, the number of tests generated for this fault in this try was set to 0 and only the time taken by simulation of old tests was charged. The "old tst." column shows the number of times a fault was detected by an old test $\boldsymbol{x}$ generated to detect a previous fault. (To be tried for a fault number $i$, test $\boldsymbol{x}$ did not have to be successful for a fault number $j$, $j < i$.). For example, value 10 for fault 3 means that in every try (out of 10), fault 3 was detected by a test generated before to detect fault 1 or 2. In the last two columns, for each fault, the average run time and average number of tests are given. The latter is computed over the tries where no old test was successful and new tests had to be generated to detect this fault. (For that reason no number of tests is given for faults 3 and 5 that were detected by old tests in all 10 tries.)

The results of Table 1 show that *TCBP* performed much better than SAT-solvers (but worse than random testing). It was able to reuse tests generated for previous faults and it was faster even without reusing tests (e.g. for fault 1).

Table 2 shows the results of testing the miter $N$ of $M$ and $M_f$ when $M$ was made up of the circuit implementing a medium bit of a 128-bit multiplier which fed an input of a 24-input AND gate. The output of this AND gate was the output of $M$. (The other 23 inputs of this AND gate were primary inputs of $M$.) The idea was to make it much harder for random tests to propagate faults to the output. The experiments indeed showed that *random testing failed on every fault* we introduced (with the threshold of $10^6$ tests per fault). Table 2 contains

**Table 1.** Testing a circuit derived from a 128 bit multiplier

| Flt. num. | *Satzilla* | *Precosat* s. | *Mxc* s. | *random testing* | | *TCBP* | | |
|---|---|---|---|---|---|---|---|---|
| | | | | times. | #tsts | old tst. | time (s.) | #tsts |
| 1 | >1h | >1h | >1h | 4.8 | 334 | 0 | 346 | 87 |
| 2 | >1h | >1h | >1h | 7.8 | 656 | 1 | 328 | 83 |
| 3 | >1h | 2,450 | >1h | 0.2 | 15 | 10 | 0.4 | n/a |
| 4 | >1h | 931 | >1h | 2.8 | 212 | 7 | 50 | 43 |
| 5 | >1h | 1,596 | 52 | 4.6 | 205 | 10 | 0.4 | n/a |

the performance of *Precosat* and *TCBP* on a subset of 17 faults (we discarded the faults that were easy for both *Precosat* and *TCBP*). The time limit was set to 5 hours. (We did not use *Satzilla* and *Mxc* because they performed much worse than *Precosat* not being able to detect the majority of faults within the time limit while *Precosat* found all faults.)

**Table 2.** Testing a 17-fault set for a circuit with arithmetic and logic components

| | *Precosat* | *TCBP* | |
|---|---|---|---|
| | time (s) | time (s) | #tests |
| total | 54,115 | 2,919 | 562 |
| average | 3,183 | 172 | 33 |
| median | 935 | 8 | 3 |

As before, we used 10 tries to generate tests for the 17-fault set. So the total, median and average values of Table 2 were computed for the averages over 10 tries. When testing fault number $i$, $i$=2,..17 we also (as in Experiment 1) checked if this fault can be detected by a test generated for a fault 1,.., $i$-1 *in the same try.* (Typically, one had to generate tests only for 3-5 faults out of 17. The other faults were detected by old tests.)

The results of Table 2 show that on the set of hard faults we used, *TCBP* was almost 20 times faster (even though it employed the same version of *Precosat* to generate boundary points). Again, this can be attributed to a) reusing old tests; b) finding a counterexample faster even when *TCBP* had to generate new tests. Reusing of old tests explains the small median values of *TCBP* for the run time and for the number of tests. For many faults, *TCBP* generated new tests in a small number of tries (if any).

## 9 Conclusions

We introduced a test generation procedure based on the TTPE framework (Treating Tests as a Proof Encoding). Given a circuit and a property, this procedure finds tests that encode mandatory fragments of *any* resolution proof that the circuit satisfies the property. These tests are extracted from boundary points, and this process does not require a proof. The successful demonstration of these ideas implies that the study of proof systems more complex than resolution and for logics more expressive than propositional logic, may lead to new methods for generating high-quality tests for both hardware and software verification.

## References

1. Abramovici, M., Breuer, M., Friedman, D.: Digital Systems Testing and Testable Design. John Wiley & Sons, Chichester (1994)
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, ch.2, pp. 19–99. North-Holland, Amsterdam (2001)
3. Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: 7th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, Las Vegas, Nevada, pp. 220–233 (1980)
4. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)
5. Goldberg, E.: Boundary points and resolution. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 147–160. Springer, Heidelberg (2009)
6. Goldberg, E.: On bridging simulation and formal verification. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 127–141. Springer, Heidelberg (2008)
7. Marques-Silva, J., Sakallah, K.: Grasp - a new search algorithm for satisfiability. In: International conference on computer-aided design, Washington, DC, USA, pp. 220–227 (1996)
8. Satpathy, M., Butler, M.J., Leuschel, M., Ramesh, S.: Automatic testing from formal specifications. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 95–113. Springer, Heidelberg (2007)
9. Tseitin, G.S.: On the complexity of derivation in the propositional calculus. In: Zapiski nauchnykh seminarov LOMI, vol. 8, pp. 234–259 (1968)
10. Alloy system, `http://alloy.mit.edu/community`
11. CProver, `http://www.cprover.org/cbmc`
12. JavaPathfinder, `http://babelfish.arc.nasa.gov/trac/jpf`
13. SAT 2009 competition, `http://www.satcompetition.org/2009`

# Relational Analysis of (Co)inductive Predicates, (Co)algebraic Datatypes, and (Co)recursive Functions[⋆]

Jasmin Christian Blanchette

Institut für Informatik, Technische Universität München, Germany
blanchette@in.tum.de

**Abstract.** This paper presents techniques for applying a finite relational model finder to logical specifications that involve (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. In contrast to previous work, which focused on algebraic datatypes and restricted occurrences of unbounded quantifiers in formulas, we can handle arbitrary formulas by means of a three-valued Kleene logic. The techniques form the basis of the counterexample generator Nitpick for Isabelle/HOL. As a case study, we consider a coalgebraic lazy list type.

## 1 Introduction

SAT and SMT solvers, model checkers, model finders, and other lightweight formal methods are today available to test or verify specifications written in various languages. These tools are often integrated in more powerful systems, such as interactive theorem provers, to discharge proof obligations or generate (counter)models.

For testing logical specifications, a particularly attractive approach is to express these in first-order relational logic (FORL) and use a model finder such as Kodkod [30] to find counterexamples. FORL extends traditional first-order logic (FOL) with relational calculus operators and the transitive closure, and offers a good compromise between automation and expressiveness. Kodkod relies on a SAT solver and forms the basis of Alloy [15]. In a case study, the Alloy Analyzer checked a mechanized version of the paper proof of the Mondex protocol and revealed several bugs in the proof [27].

However, FORL lacks the high-level definitional principles usually provided in interactive theorem provers, namely (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions (Sect. 3). Solutions have been proposed by Kuncak and Jackson [21], who modeled lists and trees in Alloy, and Dunets et al. [10], who showed how to translate algebraic datatypes and recursive functions in the context of the first-order theorem prover KIV. In both cases, the translation is restricted to formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes.

This paper generalizes previous work in several directions: First, we lift the unbounded quantifier restriction by using a three-valued logic coded in terms of the binary logic FORL (Sect. 4.2). Second, we show how to translate (co)inductive predicates, coalgebraic datatypes, and corecursive functions (Sect. 5). Third, in our treatment of algebraic datatypes, we show how to handle mutually recursive datatypes (Sect. 5.2).

The use of a three-valued Kleene logic makes it possible to analyze formulas such as $\mathit{True} \vee \forall n^{\mathit{nat}}.\, P(n)$, which are rejected by Kuncak and Jackson's syntactic criterion.

[⋆] This work is supported by the DFG grant Ni 491/11-1.

Unbounded universal quantification is still problematic in general, but suitable definitional principles and their proper handling mitigate this problem.

The ideas presented here form the basis of the higher-order counterexample generator Nitpick [5], which is included with recent versions of Isabelle/HOL [25]. As a case study, we employ Nitpick on a small theory of coalgebraic (lazy) lists (Sect. 6). To simplify the presentation, we use FOL as our specification language in the paper; issues specific to higher-order logic (HOL) are mostly orthogonal and covered elsewhere [5].

## 2 Logics

### 2.1 First-Order Logic (FOL)

The first-order logic that will serve as our specification language is essentially the first-order fragment of HOL [7,12]. The types and terms are given below.

| *Types:* | | *Terms:* | |
|---|---|---|---|
| $\sigma ::= \kappa$ | (atomic type) | $t ::= x^{\sigma}$ | (variable) |
| $\quad\mid \alpha$ | (type variable) | $\quad\mid c^{\tau}(t,\ldots,t)$ | (function term) |
| $\tau ::= (\sigma,\ldots,\sigma) \to \sigma$ | (function type) | $\quad\mid \forall x^{\sigma}.\, t$ | (universal quantification) |

The standard semantics interprets the Boolean type $o$ and the constants $\mathit{False}^{o}$, $\mathit{True}^{o}$, $\longrightarrow^{(o,o)\to o}$ (implication), $\simeq^{(\sigma,\sigma)\to o}$ (equality on basic type $\sigma$), and $\mathit{if\ then\ else}^{(o,\sigma,\sigma)\to\sigma}$. Formulas are terms of type $o$. We assume throughout this paper that terms are well-typed using the standard typing rules and usually omit the type superscripts. In conformity with first-order practice, application of $x$ and $y$ on $f$ is written $f(x,y)$, the function type $() \to \sigma$ is identified with $\sigma$, and the parentheses in the function term $c()$ are optional. We also assume that the connectives $\neg$, $\wedge$, $\vee$ and existential quantification are available.

In contrast to HOL, our logic requires variables to range over basic types, and it forbids partial function application and $\lambda$-abstractions. On the other hand, it supports the limited form of polymorphism provided by proof assistants for HOL [14,25,29], with the restriction that type variables may only be instantiated by atomic types (or left uninstantiated in a polymorphic formula).

Types and terms are interpreted in the standard set-theoretic way, relative to a scope that fixes the interpretation of basic types. A *scope* $S$ is a function from basic types to nonempty sets (domains), which need not be finite.[1] We require $S(o) = \{\mathit{ff}, \mathit{tt}\}$.

The standard interpretation $[\![\tau]\!]_S$ of a type $\tau$ is given by

$$[\![\sigma]\!]_S = S(\sigma) \qquad [\![(\sigma_1,\ldots,\sigma_n) \to \sigma]\!]_S = [\![\sigma_1]\!]_S \times \cdots \times [\![\sigma_n]\!]_S \to [\![\sigma]\!]_S,$$

where $A \to B$ denotes the set of (total) functions from $A$ to $B$. In contexts where $S$ is clear, the cardinality of $[\![\tau]\!]_S$ is written $|\tau|$.

### 2.2 First-Order Relational Logic (FORL)

Our analysis logic, first-order relational logic, combines elements from FOL and relational calculus extended with the transitive closure [15,30]. Formulas involve variables and terms ranging over relations (sets of tuples drawn from a universe of atoms) of arbitrary arities. The logic is unsorted, but each term denotes a relation of a fixed arity.

[1] The use of the word "scope" for a domain specification is consistent with Jackson [15].

*Formulas:*

| | | |
|---|---|---|
| $\varphi ::=$ | $\mathsf{false}$ | (falsity) |
| | $\mid \mathsf{true}$ | (truth) |
| | $\mid m\ r$ | (multiplicity constraint) |
| | $\mid r \simeq r$ | (equality) |
| | $\mid r \subseteq r$ | (inclusion) |
| | $\mid \neg\varphi$ | (negation) |
| | $\mid \varphi \wedge \varphi$ | (conjunction) |
| | $\mid \forall x{\in}r\colon \varphi$ | (universal quantification) |

$m ::= \mathsf{no} \mid \mathsf{lone} \mid \mathsf{one} \mid \mathsf{some}$

*Terms:*

| | | |
|---|---|---|
| $r ::=$ | $\mathsf{none}$ | (empty set) |
| | $\mid \mathsf{iden}$ | (identity relation) |
| | $\mid \mathsf{a}_i$ | (atom) |
| | $\mid x$ | (variable) |
| | $\mid r^+$ | (transitive closure) |
| | $\mid r.r$ | (dot-join) |
| | $\mid r \times r$ | (Cartesian product) |
| | $\mid r \cup r$ | (union) |
| | $\mid r - r$ | (difference) |
| | $\mid \mathsf{if}\ \varphi\ \mathsf{then}\ r\ \mathsf{else}\ r$ | (conditional) |

The universe of discourse is $\mathscr{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$, where each $\mathsf{a}_i$ is an uninterpreted atom. Atoms and $n$-tuples are identified with singleton sets and singleton $n$-ary relations, respectively. Bound variables in quantifications range over the tuples in a relation; thus, $\forall x \in (\mathsf{a}_1 \cup \mathsf{a}_2) \times \mathsf{a}_3\colon \varphi(x)$ is equivalent to $\varphi(\mathsf{a}_1 \times \mathsf{a}_3) \wedge \varphi(\mathsf{a}_2 \times \mathsf{a}_3)$.

Although they are not listed above, we will sometimes make use of $\vee$, $\longrightarrow$, $^*$, and $\cap$ as well. The constraint $\mathsf{no}\ r$ expresses that $r$ is the empty relation, $\mathsf{one}\ r$ expresses that $r$ is a singleton, $\mathsf{lone}\ r \iff \mathsf{no}\ r \vee \mathsf{one}\ r$, and $\mathsf{some}\ r \iff \neg\,\mathsf{no}\ r$. The dot-join operator is unconventional; its semantics is given by the equation

$$[\![r.s]\!] = \{(r_1, \ldots, r_{m-1}, s_2, \ldots, s_n) \mid \exists t.\ (r_1, \ldots, r_{m-1}, t) \in [\![r]\!] \wedge (t, s_2, \ldots, s_n) \in [\![s]\!]\}.$$

The operator admits three important special cases. Let $s$ be unary and $r$, $r'$ be binary relations. The expression $s.r$ gives the direct image of the set $s$ under $r$; if $s$ is a singleton and $r$ a function, it coincides with the function application $r(s)$. Analogously, $r.s$ gives the inverse image of $s$ under $r$. Finally, $r.r'$ expresses relational composition.

The following FORL specification attempts to fit 30 pigeons in 29 holes:

$$\begin{array}{l} \textbf{vars}\ \mathit{pigeons} = \{\mathsf{a}_1, \ldots, \mathsf{a}_{30}\},\ \mathit{holes} = \{\mathsf{a}_{31}, \ldots, \mathsf{a}_{59}\} \\ \textbf{var}\ \emptyset \subseteq \mathit{nest} \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_{30}\} \times \{\mathsf{a}_{31}, \ldots, \mathsf{a}_{59}\} \\ \textbf{solve}\ (\forall p \in \mathit{pigeons}\colon \mathsf{one}\ p.\mathit{nest}) \wedge (\forall h \in \mathit{holes}\colon \mathsf{lone}\ \mathit{nest}.h) \end{array}$$

The variables *pigeons* and *holes* are given fixed values, whereas *nest* is specified with a lower and an upper bound. The constraint $\mathsf{one}\ p.\mathit{nest}$ states that pigeon $p$ is in relation with exactly one hole, and $\mathsf{lone}\ \mathit{nest}.h$ that hole $h$ is in relation with at most one pigeon. Taken as a whole, the formula states that *nest* is a one-to-one function. It is, of course, not satisfiable, a fact that Kodkod can establish in less than a second.

## 3 Definitional Principles

### 3.1 Simple Definitions

We extend our specification logic FOL with several definitional principles to introduce new constants and types. The first principle defines a constant as equal to another term:

$$\textbf{definition}\ c^{\tau}\ \textbf{where}\ c(\bar{x}) \simeq t$$

Logically, the above definition is equivalent to the axiom $\forall \bar{x}.\ c(\bar{x}) \simeq t$.

Provisos: The constant $c$ is fresh, the variables $\bar{x}$ are distinct, and the right-hand side $t$ does not refer to any other free variables than $\bar{x}$, to any undefined constants or $c$, or to any type variables not occurring in $\tau$. These restrictions ensure consistency [32].

### 3.2 (Co)inductive Predicates

The **inductive** and **coinductive** commands define inductive and coinductive predicates specified by their introduction rules:

$$\begin{array}{l}
[\textbf{co}]\textbf{inductive}\ p^{\tau}\ \textbf{where} \\
p(\bar{t}_{11}) \wedge \cdots \wedge p(\bar{t}_{1\ell_1}) \wedge Q_1 \longrightarrow p(\bar{u}_1) \\
\quad\vdots \\
p(\bar{t}_{n1}) \wedge \cdots \wedge p(\bar{t}_{n\ell_n}) \wedge Q_n \longrightarrow p(\bar{u}_n)
\end{array}$$

Provisos: The constant $p$ is fresh, and the arguments to $p$ and the side conditions $Q_i$ do not refer to $p$, undeclared constants, or any type variables not occurring in $\tau$.

The introduction rules may involve any number of free variables $\bar{y}$. The syntactic restrictions on the rules ensure monotonicity; by the Knaster–Tarski theorem, the fixed point equation

$$p(\bar{x}) \simeq \exists\bar{y}.\ \bigvee_{j=1}^{n} \bar{x} \simeq \bar{u}_j \wedge p(\bar{t}_{j1}) \wedge \cdots \wedge p(\bar{t}_{j\ell_j}) \wedge Q_j$$

admits a least and a greatest solution [13, 26]. Inductive definitions provide the least fixed point, and coinductive definitions provide the greatest fixed point.

As an example, assuming a type *nat* of natural numbers generated freely by $0^{nat}$ and $Suc^{nat\to nat}$, the following definition introduces the predicate *even* of even numbers:

$$\begin{array}{l}
\textbf{inductive}\ even^{nat\to o}\ \textbf{where} \\
even(0) \\
even(n) \longrightarrow even(Suc(Suc(n)))
\end{array}$$

The associated fixed point equation is

$$even(x) \simeq \exists n.\ x \simeq 0 \vee x \simeq Suc(Suc(n)) \wedge even(n).$$

The syntax can be generalized to support mutual definitions, as in the next example:

$$\begin{array}{l}
\textbf{inductive}\ even^{nat\to o}\ \textbf{and}\ odd^{nat\to o}\ \textbf{where} \\
even(0) \\
even(n) \longrightarrow odd(Suc(n)) \\
odd(n) \longrightarrow even(Suc(n))
\end{array}$$

Mutual definitions for $p_1, \ldots, p_m$ can be reduced to a single predicate $q$ whose domain is the disjoint sum of the domains of each $p_i$ [26]. Assuming *Inl* and *Inr* are the disjoint sum constructors, the definition of *even* and *odd* is replaced by

$$\begin{array}{l}
\textbf{inductive}\ even\_or\_odd^{(nat,nat)\,sum\to o}\ \textbf{where} \\
even\_or\_odd(Inl(0)) \\
even\_or\_odd(Inl(n)) \longrightarrow even\_or\_odd(Inr(Suc(n))) \\
even\_or\_odd(Inr(n)) \longrightarrow even\_or\_odd(Inl(Suc(n)))
\end{array}$$

$$\begin{array}{l}
\textbf{definition}\ even^{nat\to o}\ \textbf{where}\ even(n) \simeq even\_or\_odd(Inl(n)) \\
\textbf{definition}\ odd^{nat\to o}\ \textbf{where}\ odd(n) \simeq even\_or\_odd(Inr(n))
\end{array}$$

### 3.3 (Co)algebraic Datatypes

The **datatype** and **codatatype** commands define mutually recursive (co)algebraic datatypes specified by their constructors:

$$\begin{aligned}&[\textbf{co}]\textbf{datatype}\ (\bar{\alpha})\,\kappa_1 = C_{11}\ \left[\textbf{of}\ \bar{\sigma}_{11}\right]\ \mid\ \cdots\ \mid\ C_{1\ell_1}\ \left[\textbf{of}\ \bar{\sigma}_{1\ell_1}\right]\\ &\qquad\textbf{and}\ \ldots\\ &\qquad\textbf{and}\ (\bar{\alpha})\,\kappa_n = C_{n1}\ \left[\textbf{of}\ \bar{\sigma}_{n1}\right]\ \mid\ \cdots\ \mid\ C_{n\ell_n}\ \left[\textbf{of}\ \bar{\sigma}_{n\ell_n}\right]\end{aligned}$$

The defined types $(\bar{\alpha})\,\kappa_i$ are parameterized by a list of distinct type variables $\bar{\alpha}$, providing type polymorphism. Each constructor $C_{ij}$ has type $\bar{\sigma}_{ij} \to (\bar{\alpha})\,\kappa_i$.

Provisos: The type names $\kappa_i$ and the constructor constants $C_{ij}$ are fresh and distinct, the type parameters $\bar{\alpha}$ are distinct, and the argument types $\bar{\sigma}_{ij}$ do not refer to any other type variables than $\bar{\alpha}$ (but may refer to the types $(\bar{\alpha})\,\kappa_i$ being defined).

The commands can be used to define natural numbers, pairs, finite lists, and possibly infinite lazy lists as follows:

$$\begin{array}{ll}\textbf{datatype}\ \mathit{nat} = 0 \mid \mathit{Suc}\ \textbf{of}\ \mathit{nat} & \textbf{datatype}\ \alpha\,\mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ \textbf{of}\ (\alpha, \alpha\,\mathit{list})\\ \textbf{datatype}\ (\alpha,\beta)\,\mathit{pair} = \mathit{Pair}\ \textbf{of}\ (\alpha,\beta) & \textbf{codatatype}\ \alpha\,\mathit{llist} = \mathit{LNil} \mid \mathit{LCons}\ \textbf{of}\ (\alpha, \alpha\,\mathit{llist})\end{array}$$

Defining a (co)datatype introduces the appropriate axioms for the constructors [26]. It also introduces the syntax $\mathit{case}\ t\ \mathit{of}\ C_{i1}(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_{i\ell_i}(\bar{x}_{\ell_i}) \Rightarrow u_{\ell_i}$, characterized by $\forall \bar{x}_j.\ (\mathit{case}\ C_{ij}(\bar{x}_j)\ \mathit{of}\ C_{i1}(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_{i\ell_i}(\bar{x}_{\ell_i}) \Rightarrow u_{\ell_i}) \simeq u_j$ for $j \in \{1,\ldots,\ell_i\}$.

### 3.4 (Co)recursive Functions

The **primrec** command defines primitive recursive functions on algebraic datatypes:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{primrec}\ f_1^{\tau_1}\ \textbf{and}\ \ldots\ \textbf{and}\ f_n^{\tau_n}\ \textbf{where}}\\ f_1(C_{11}(\bar{x}_{11}), \bar{z}_{11}) \simeq t_{11} & \ldots & f_1(C_{1\ell_1}(\bar{x}_{1\ell_1}), \bar{z}_{1\ell_1}) \simeq t_{1\ell_1}\\ \quad\vdots & & \\ f_n(C_{n1}(\bar{x}_{n1}), \bar{z}_{n1}) \simeq t_{n1} & \ldots & f_n(C_{n\ell_n}(\bar{x}_{n\ell_n}), \bar{z}_{n\ell_n}) \simeq t_{n\ell_n}\end{array}$$

Provisos: The constants $f_i$ are fresh and distinct, the variables $\bar{x}_{ij}$ and $\bar{z}_{ij}$ are distinct for any given $i$ and $j$, the right-hand sides $t_{ij}$ involve no other variables than $\bar{x}_{ij}$ and $\bar{z}_{ij}$ and no type variables that do not occur in $\tau_i$, and the first argument of any recursive call must be one of the $\bar{x}_{ij}$'s. The recursion is well-founded because each recursive call peels off one constructor from the first argument.

Corecursive function definitions follow a rather different syntactic schema, with a single equation per function $f_i$ that must return type $(\bar{\alpha})\,\kappa_i$:

$$\begin{array}{l}\textbf{coprimrec}\ f_1^{\tau_1}\ \textbf{and}\ \ldots\ \textbf{and}\ f_n^{\tau_n}\ \textbf{where}\\ f_1(\bar{y}_1) \simeq \mathit{if}\ Q_{11}\ \mathit{then}\ t_{11}\ \mathit{else\ if}\ Q_{12}\ \mathit{then}\ \ldots\ \mathit{else}\ t_{1\ell_1}\\ \quad\vdots\\ f_n(\bar{y}_n) \simeq \mathit{if}\ Q_{n1}\ \mathit{then}\ t_{n1}\ \mathit{else\ if}\ Q_{n2}\ \mathit{then}\ \ldots\ \mathit{else}\ t_{n\ell_n}\end{array}$$

Provisos: The constants $f_i$ are fresh and distinct, the variables $\bar{y}_i$ are distinct, the right-hand sides involve no other variables than $\bar{y}_i$, no corecursive calls occur in the conditions

$\bar{Q}_{ij}$, and either $\bar{t}_{ij}$ does not involve any corecursive calls or it has the form $\bar{C}_{ij}(\bar{u}_{ij})$.[2] The syntax can be relaxed to allow a *case* expression instead of a sequence of conditionals.

The following examples define concatenation for $\alpha$ *list* and $\alpha$ *llist*:

$$\textbf{primrec}\ cat^{(\alpha\,list,\alpha\,list)\to\alpha\,list}\ \textbf{where}$$
$$cat(Nil, zs) \simeq zs \qquad cat(Cons(y, ys), zs) \simeq Cons(y, cat(ys, zs))$$

$$\textbf{coprimrec}\ lcat^{(\alpha\,list,\alpha\,list)\to\alpha\,list}\ \textbf{where}$$
$$lcat(ys, zs) \simeq case\ ys\ of\ LNil \Rightarrow zs \mid LCons(y, ys') \Rightarrow LCons(y, lcat(ys', zs))$$

## 4 Basic Translations

### 4.1 A Sound and Complete Translation

This section presents the translation of FOL to FORL, excluding the definitional principles from Sect. 3. We consider only finite domains; for these the translation is sound and complete. We start by mapping FOL types $\tau$ to sets of FORL atom tuples $\langle\!\langle \tau \rangle\!\rangle$:

$$\langle\!\langle \sigma \rangle\!\rangle = \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\sigma|}\} \qquad \langle\!\langle (\sigma_1, \ldots, \sigma_n) \to \sigma \rangle\!\rangle = \langle\!\langle \sigma_1 \rangle\!\rangle \times \cdots \times \langle\!\langle \sigma_n \rangle\!\rangle \times \langle\!\langle \sigma \rangle\!\rangle.$$

For simplicity, we reuse the same atoms for distinct basic types. A real implementation can benefit from using distinct atoms because it facilitates symmetry breaking [30].

For each free variable or nonstandard constant $u^\tau$, we generate the bounds declaration $\textbf{var}\ \emptyset \subseteq u \subseteq \langle\!\langle \tau \rangle\!\rangle$ as well as a constraint $\Phi(u)$ to ensure that single values are singletons and functions are functions:

$$\Phi(u^\sigma) = \mathsf{one}\ u \qquad \Phi(u^{(\varsigma_1,\ldots,\varsigma_n)\to\varsigma}) = \forall x_1 \in \langle\!\langle \varsigma_1 \rangle\!\rangle, \ldots, x_n \in \langle\!\langle \varsigma_n \rangle\!\rangle\colon \mathsf{one}\ x_n.(\ldots.(x_1.u)\ldots).$$

Since FORL distinguishes between formulas and terms, the translation to FORL is performed by two mutually recursive functions, $\mathsf{F}\langle\!\langle t \rangle\!\rangle$ and $\mathsf{T}\langle\!\langle t \rangle\!\rangle$:[3]

$$\begin{aligned}
\mathsf{F}\langle\!\langle False \rangle\!\rangle &= \mathsf{false} & \mathsf{T}\langle\!\langle x \rangle\!\rangle &= x\\
\mathsf{F}\langle\!\langle True \rangle\!\rangle &= \mathsf{true} & \mathsf{T}\langle\!\langle False \rangle\!\rangle &= \mathsf{a}_1\\
\mathsf{F}\langle\!\langle t \simeq u \rangle\!\rangle &= \mathsf{T}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle u \rangle\!\rangle & \mathsf{T}\langle\!\langle True \rangle\!\rangle &= \mathsf{a}_2\\
\mathsf{F}\langle\!\langle t \longrightarrow u \rangle\!\rangle &= \mathsf{F}\langle\!\langle t \rangle\!\rangle \longrightarrow \mathsf{F}\langle\!\langle u \rangle\!\rangle & \mathsf{T}\langle\!\langle if\ t\ then\ u_1\ else\ u_2 \rangle\!\rangle &= \mathsf{if}\ \mathsf{F}\langle\!\langle t \rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1 \rangle\!\rangle\ \mathsf{else}\ \mathsf{T}\langle\!\langle u_2 \rangle\!\rangle\\
\mathsf{F}\langle\!\langle \forall x^\sigma.\, t \rangle\!\rangle &= \forall x \in \langle\!\langle \sigma \rangle\!\rangle\colon \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{T}\langle\!\langle c(t_1, \ldots, t_n) \rangle\!\rangle &= \mathsf{T}\langle\!\langle t_n \rangle\!\rangle.(\ldots.(\mathsf{T}\langle\!\langle t_1 \rangle\!\rangle.c)\ldots)\\
\mathsf{F}\langle\!\langle t \rangle\!\rangle &= \mathsf{T}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle True \rangle\!\rangle & \mathsf{T}\langle\!\langle t^o \rangle\!\rangle &= \mathsf{T}\langle\!\langle if\ t\ then\ True\ else\ False \rangle\!\rangle.
\end{aligned}$$

The metavariable $c$ ranges over nonstandard constants, so that the $\mathsf{T}\langle\!\langle t^o \rangle\!\rangle$ equation is used for $\simeq$ and $\longrightarrow$ (as well as for $\forall$). The Boolean values false and true are arbitrarily coded as $\mathsf{a}_1$ and $\mathsf{a}_2$ when they appear as FORL terms.

**Theorem 4.1.** *The FOL formula $P$ with free variables and nonstandard constants $u_1^{\tau_1}$, …, $u_n^{\tau_n}$ is satisfiable for a given finite scope iff the FORL formula $\mathsf{F}\langle\!\langle P \rangle\!\rangle \wedge \bigwedge_{j=1}^{n} \Phi(u_j)$ with bounds $\emptyset \subseteq u_j \subseteq \langle\!\langle \tau_j \rangle\!\rangle$ is satisfiable for the same scope.*

[2] Other authors formulate corecursion in terms of selectors instead of constructors [16].

[3] Metatheoretic functions here and elsewhere are defined using sequential pattern matching.

*Proof.* Let $[\![t]\!]_M$ denote the set-theoretic semantics of the FOL term $t$ w.r.t. a model $M$ and the given scope $S$, let $[\![\varphi]\!]_V$ denote the truth value of the FORL formula $\varphi$ w.r.t. a variable valuation $V$ and the scope $S$, and let $[\![r]\!]_V$ denote the set-theoretic semantics of the FORL term $r$ w.r.t. $V$ and $S$. Furthermore, for $v \in [\![\sigma]\!]_S$, let $\lfloor v \rfloor$ denote the corresponding value in $\langle\!\langle \sigma \rangle\!\rangle$, with $\lfloor \mathit{ff} \rfloor = \mathsf{a}_1$ and $\lfloor \mathit{tt} \rfloor = \mathsf{a}_2$. Using recursion induction, it is straightforward to prove that $[\![\mathsf{F}\langle\!\langle t^o \rangle\!\rangle]\!]_V \iff [\![t]\!]_M = \mathit{tt}$ and $[\![\mathsf{T}\langle\!\langle t \rangle\!\rangle]\!]_V = \lfloor [\![t]\!]_M \rfloor$ if $V(u_j) = \lfloor M(u_j) \rfloor$ for all $u_j$'s. Moreover, from a satisfying valuation $V$ of the $u_j$'s, we can construct a FOL model $M$ such that $\lfloor M(u_j) \rfloor = V(u_j)$; the $\Phi$ constraints and the bounds ensure that such a model exists. Hence, $[\![\mathsf{F}\langle\!\langle P \rangle\!\rangle]\!]_V \iff [\![P]\!]_M = \mathit{tt}$. □

The translation is parameterized by a scope, which specifies the exact cardinalities of the basic types occurring in the formula. To exhaust all models up to a cardinality bound $k$ for $n$ basic types, a model finder must a priori iterate through $k^n$ combinations of cardinalities and must consider all models for each of these combinations. This can be made more efficient by taking the cardinalities as upper bounds rather than exact bounds (Alloy's default mode of operation [15, p. 129]) or by inferring scope monotonicity [4,21].

## 4.2 Approximation of Infinite Types and Partiality

Besides its lack of support for the definitional principles, the above translation suffers from a serious limitation: It disregards infinite types such as natural numbers, lists, and trees, which are ubiquitous in real-world specifications. Fortunately, it is not hard to adapt the translation to take these into account in a sound (but incomplete) way.

Given an infinite atomic type $\kappa$, we consider a finite subset of $[\![\kappa]\!]_S$ and map every element not in this subset to a special undefined value $\perp$. For the type *nat* of natural numbers, an obvious choice is to consider prefixes $\{0, \ldots, K\}$ of $\mathbb{N}$ and map numbers $> K$ to $\perp$. Observe that the successor function *Suc* becomes partial, with $\mathit{Suc}\ K = \perp$. The technique can also be used to speed up the analysis of finite types with a high cardinality: We can approximate a 256-value *byte* type by a subset of, say, 5 values.

Leaving out some elements of atomic types means that we must cope with partiality. Not only may functions be partial, but any term or formula can evaluate to $\perp$. The logic becomes a three-valued Kleene logic [17]. Universal quantifiers whose bound variable ranges over an approximated type, such as $\forall n^{\mathit{nat}}.\ P(n)$, will evaluate to either *False* (if $P(n)$ gives *False* for some $n \leq K$) or $\perp$, but never to *True*, since we do not know whether $P(K+1)$, $P(K+2)$, $\ldots$, are true.

Partiality can be encoded in FORL as follows. Inside terms, we let $\mathsf{none}$ (the empty set) stand for $\perp$. This choice is convenient because $\mathsf{none}$ is an absorbing element for the dot-join operator, which models function application; thus, $f(\perp) = \perp$. Inside a formula, we keep track of the polarity of the subformulas: In positive contexts (i.e., under an even number of negations), $\mathsf{true}$ codes *True* and $\mathsf{false}$ codes *False* or $\perp$; in negative contexts, $\mathsf{false}$ codes *False* and $\mathsf{true}$ codes *True* or $\perp$.

The translation of FOL terms is performed by two functions, $\mathsf{F}^s\langle\!\langle t \rangle\!\rangle$ and $\mathsf{T}\langle\!\langle t \rangle\!\rangle$, where $s$ indicates the polarity ($+$ or $-$):

$$\mathsf{F}^s\langle\!\langle \mathit{False} \rangle\!\rangle = \mathsf{false} \qquad\qquad \mathsf{T}\langle\!\langle x \rangle\!\rangle = x$$

$$\mathsf{F}^s\langle\!\langle \mathit{True} \rangle\!\rangle = \mathsf{true} \qquad\qquad \mathsf{T}\langle\!\langle \mathit{False} \rangle\!\rangle = \mathsf{a}_1$$

$$\begin{aligned}
\mathsf{F}^+\langle\!\langle t \simeq u\rangle\!\rangle &= \mathsf{some}\ (\mathsf{T}\langle\!\langle t\rangle\!\rangle \cap \mathsf{T}\langle\!\langle u\rangle\!\rangle) \\
\mathsf{F}^-\langle\!\langle t \simeq u\rangle\!\rangle &= \mathsf{lone}\ (\mathsf{T}\langle\!\langle t\rangle\!\rangle \cup \mathsf{T}\langle\!\langle u\rangle\!\rangle) \\
\mathsf{F}^s\langle\!\langle t \longrightarrow u\rangle\!\rangle &= \mathsf{F}^{-s}\langle\!\langle t\rangle\!\rangle \longrightarrow \mathsf{F}^s\langle\!\langle u\rangle\!\rangle \\
\mathsf{F}^+\langle\!\langle \forall x^\sigma\!.\, t\rangle\!\rangle &= \mathsf{false} \quad \text{if } |\langle\!\langle \sigma\rangle\!\rangle| < |\sigma| \\
\mathsf{F}^s\langle\!\langle \forall x^\sigma\!.\, t\rangle\!\rangle &= \forall x \in \langle\!\langle \sigma\rangle\!\rangle \colon \mathsf{F}^s\langle\!\langle t\rangle\!\rangle \\
\mathsf{F}^+\langle\!\langle t\rangle\!\rangle &= \mathsf{T}\langle\!\langle t\rangle\!\rangle \simeq \mathsf{T}\langle\!\langle \mathit{True}\rangle\!\rangle \\
\mathsf{F}^-\langle\!\langle t\rangle\!\rangle &= \mathsf{T}\langle\!\langle t\rangle\!\rangle \not\simeq \mathsf{T}\langle\!\langle \mathit{False}\rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{T}\langle\!\langle \mathit{True}\rangle\!\rangle &= \mathsf{a}_2 \\
\mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ u_1\ \mathit{else}\ u_2\rangle\!\rangle &= \mathsf{if}\ \mathsf{F}^+\langle\!\langle t\rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1\rangle\!\rangle \\
&\phantom{=}\ \mathsf{else\ if}\ \neg\,\mathsf{F}^-\langle\!\langle t\rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_2\rangle\!\rangle \\
&\phantom{=}\ \mathsf{else\ none} \\
\mathsf{T}\langle\!\langle c(t_1,\ldots,t_n)\rangle\!\rangle &= \mathsf{T}\langle\!\langle t_n\rangle\!\rangle.(\ldots.(\mathsf{T}\langle\!\langle t_1\rangle\!\rangle.c)\ldots) \\
\mathsf{T}\langle\!\langle t^o\rangle\!\rangle &= \mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ \mathit{True} \\
&\phantom{=}\ \mathit{else}\ \mathit{False}\rangle\!\rangle.
\end{aligned}$$

In the equation for implication, $-s$ denotes $-$ if $s$ is $+$ and $+$ if $s$ is $-$. Taken together, $(\mathsf{F}^+\langle\!\langle t\rangle\!\rangle, \mathsf{F}^-\langle\!\langle t\rangle\!\rangle)$ encode a three-valued logic, with $(\mathsf{true}, \mathsf{true})$ corresponding to *True*, $(\mathsf{false}, \mathsf{true})$ corresponding to $\bot$, and $(\mathsf{false}, \mathsf{false})$ corresponding to *False*. The case $(\mathsf{true}, \mathsf{false})$ is impossible by construction.

When mapping FOL types to sets of FORL atom tuples, basic types $\sigma$ are now allowed to take any finite cardinality $|\langle\!\langle \sigma\rangle\!\rangle| \le |\sigma|$. We also need to relax the definition of $\Phi(u)$ to allow empty sets, by substituting $\mathsf{lone}$ for $\mathsf{one}$.

**Theorem 4.2.** *Given a FOL formula $P$ with free variables and nonstandard constants $u_1^{\tau_1}, \ldots, u_n^{\tau_n}$ and a scope $S$, the FORL formula $\mathsf{F}^+\langle\!\langle P\rangle\!\rangle \wedge \bigwedge_{j=1}^n \Phi(u_j)$ with bounds $\emptyset \subseteq u_j \subseteq \langle\!\langle \tau_j\rangle\!\rangle$ is satisfiable for $S$ only if $P$ is satisfiable for $S$.*

*Proof.* The proof is similar to that of Theorem 4.1, but partiality requires us to compare the actual value of a FORL expression with its expected value using $\subseteq$ rather than $=$. Using recursion induction, we can prove that $[\![\mathsf{F}^+\langle\!\langle t^o\rangle\!\rangle]\!]_V \Longrightarrow [\![t]\!]_M = \mathit{tt}$, $\neg\,[\![\mathsf{F}^-\langle\!\langle t^o\rangle\!\rangle]\!]_V \Longrightarrow [\![t]\!]_M = \mathit{ff}$, and $[\![\mathsf{T}\langle\!\langle t\rangle\!\rangle]\!]_V \subseteq \lfloor [\![t]\!]_M \rfloor$ if $V(u) \subseteq \lfloor M(u) \rfloor$ for all free variables and nonstandard constants $u$ occurring in $t$. Some of the cases deserve more justification:

- The $\mathsf{F}^+\langle\!\langle t \simeq u\rangle\!\rangle$ equation is sound because if the intersection of $\mathsf{T}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{T}\langle\!\langle u\rangle\!\rangle$ is nonempty, then $t$ and $u$ must be equal (since they are singletons).
- The $\mathsf{F}^-\langle\!\langle t \simeq u\rangle\!\rangle$ equation is dual: If the union of $\mathsf{T}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{T}\langle\!\langle u\rangle\!\rangle$ has more than one element, then $t$ and $u$ must be unequal.
- Universal quantification occurring positively can never yield true if the bound variable ranges over an approximated type. (In negative contexts, approximation compromises the encoding's completeness but not its soundness.)
- The *if then else* equation carefully distinguishes between the cases where the condition is *True*, *False*, and $\bot$. In the *True* case, it returns the *then* value; in the *False* case, it returns the *else* value; and in the $\bot$ case, it returns $\bot$ ($\mathsf{none}$).
- The $\mathsf{T}\langle\!\langle c(t_1,\ldots,t_n)\rangle\!\rangle$ equation is as before. If any of the arguments $t_j$ evaluates to $\mathsf{none}$, the entire dot-join expression yields $\mathsf{none}$.

Moreover, from a satisfying valuation $V$ of the $u_j$'s, we can construct a FOL model $M$ such that $V(u_j) \subseteq \lfloor M(u_j) \rfloor$ for all $u_j$'s, by defining $M(u_j)$ arbitrarily if $V(u_j) = \emptyset$ or at points where the partial function $V(u_j)$ is undefined. Hence, $[\![\mathsf{F}^+\langle\!\langle P\rangle\!\rangle]\!]_V$ implies $[\![P]\!]_M = \mathit{tt}$. □

Although our translation is sound, a lot of precision is lost for $\simeq$ and $\forall$. Fortunately, by handling high-level definitional principles specially (as opposed to directly translating their FOL axiomatization), we can bypass the imprecise translation and increase the precision. This is covered in the next section.

## 5 Translation of Definitional Principles

### 5.1 Axiomatization of Simple Definitions

Once we extend the specification logic with simple definitions, we must also encode these in the FORL formula. More precisely, if $c^{\tau}$ is defined and an instance $c^{\tau'}$ occurs in a formula, we must conjoin $c$'s definition with the formula, instantiating $\tau$ with $\tau'$. This process must be repeated for any defined constants occurring in $c$'s definition.

Given the command

$$\textbf{definition } c^{\tau} \textbf{ where } c(\bar{x}) \simeq t$$

the naive approach would be to conjoin $\mathsf{F}^{+}\langle\!\langle \forall \bar{x}.\ c(\bar{x}) \simeq t \rangle\!\rangle$ with the FORL formula to satisfy and recursively do the same for any defined constants in $t$. However, there are two problems with this approach:

– If any of the variables $\bar{x}$ is of an approximated type, the equation $\mathsf{F}^{+}\langle\!\langle \forall \bar{x}.\ t \rangle\!\rangle = \mathsf{false}$ applies, and the axiom becomes unsatisfiable. This is sound but extremely imprecise, as it prevents the discovery of any model.
– Otherwise, the body of $\forall \bar{x}.\ c(\bar{x}) \simeq t$ is translated to some $(\mathsf{T}\langle\!\langle c(\bar{x}) \rangle\!\rangle \cap \mathsf{T}\langle\!\langle t \rangle\!\rangle)$, which evaluates to $\mathsf{false}$ whenever $\mathsf{T}\langle\!\langle t \rangle\!\rangle$ is $\mathsf{none}$ for some values of $\bar{x}$.

Fortunately, we can take a shortcut and translate the definition directly to the following FORL axiom, bypassing $\mathsf{F}^{+}$ altogether (cf. Weber [31, p. 66]):

$$\forall x_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_n \in \langle\!\langle \sigma_n \rangle\!\rangle\colon\ \mathsf{T}\langle\!\langle c(x_1, \ldots, x_n) \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t \rangle\!\rangle.$$

**Theorem 5.1.** *The encoding of Sect. 4.2 extended with simple definitions is sound.*

*Proof.* Any FORL valuation $V$ that satisfies the FORL axiom for a constant $c$ can be extended into a FOL model $M$ that satisfies the corresponding FOL axiom, by setting $M(c)(\bar{v}) = [\![t]\!]_M(\bar{v})$ for any values $\bar{v}$ at which $V(c)$ is not defined (either because $\bar{v}$ is not representable in FORL or because the partial function $V(c)$ is not defined at that point). The apparent circularity in $M(c)(\bar{v}) = [\![t]\!]_M(\bar{v})$ is harmless, because simple definitions are required to be acyclic and so we can construct $M$ one constant at a time. □

### 5.2 Axiomatization of Algebraic Datatypes and Recursive Functions

The FORL axiomatization of algebraic datatypes follows the lines of Kuncak and Jackson [21]. Let $\kappa = C_1$ **of** $(\sigma_{11}, \ldots, \sigma_{1n_1}) \mid \cdots \mid C_\ell$ **of** $(\sigma_{\ell 1}, \ldots, \sigma_{\ell n_\ell})$ be a datatype instance. With each constructor $C_i$, we associate a discriminator $D_i^{\kappa \to o}$ and $n$ selectors $S_{ik}^{\kappa \to \sigma_k}$ obeying $D_j(C_i(\bar{x})) \simeq (i \simeq j)$ and $S_{ik}(C_i(x_1, ..., x_n)) \simeq x_k$. For example, the type $\alpha$ *list* is assigned the discriminators *nilp* and *consp* and the selectors *head* and *tail*:[4]

$$\begin{array}{ccc} \mathit{nilp}(\mathit{Nil}) \simeq \mathit{True} & \mathit{nilp}(\mathit{Cons}(x, \mathit{xs})) \simeq \mathit{False} & \mathit{head}(\mathit{Cons}(x, \mathit{xs})) \simeq x \\ \mathit{consp}(\mathit{Nil}) \simeq \mathit{False} & \mathit{consp}(\mathit{Cons}(x, \mathit{xs})) \simeq \mathit{True} & \mathit{tail}(\mathit{Cons}(x, \mathit{xs})) \simeq \mathit{xs}. \end{array}$$

[4] These names were chosen for readability; any fresh names would do.

The discriminator and selector view almost always results in a more efficient SAT encoding than the constructor view because it breaks high-arity constructors into several low-arity discriminators and selectors, declared as follows (for all possible $i$, $k$):

$$\textbf{var } \emptyset \subseteq D_i \subseteq \langle\!\langle \kappa \rangle\!\rangle \qquad\qquad \textbf{var } \emptyset \subseteq S_{ik} \subseteq \langle\!\langle \kappa \to \sigma_{ik} \rangle\!\rangle$$

The predicate $D_i$ is directly coded as a set of atoms, rather than as a function to $\{\mathsf{a}_1, \mathsf{a}_2\}$.

Let $C_i\langle r_1, \ldots, r_n\rangle$ stand for $S_{i1} . r_1 \cap \cdots \cap S_{in} . r_n$ if $n \geq 1$, and $C_i\langle\rangle = D_i$ for parameterless constructors. Intuitively, $C_i\langle r_1, \ldots, r_n\rangle$ represents the constructor $C_i$ with arguments $r_1, \ldots, r_n$ at the FORL level [10]. A faithful axiomatization of datatypes in terms of $D_i$ and $S_{ik}$ involves the following axioms (for all possible $i$, $j$, $k$):

$$\begin{aligned}
\textsc{Disjoint}_{ij}\text{:}\quad & \mathsf{no}\ D_i \cap D_j \quad \text{for } i < j \\
\textsc{Exhaustive}\text{:}\quad & D_1 \cup \cdots \cup D_\ell \simeq \langle\!\langle \kappa \rangle\!\rangle \\
\textsc{Selector}_{ik}\text{:}\quad & \forall y \in \langle\!\langle \kappa \rangle\!\rangle\text{: } \mathsf{if}\ y \subseteq D_i\ \mathsf{then\ one}\ y . S_{ik}\ \mathsf{else\ no}\ y . S_{ik} \\
\textsc{Unique}_i\text{:}\quad & \forall x_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \mathsf{lone}\ C_i\langle x_1, \ldots, x_{n_i}\rangle \\
\textsc{Generator}_i\text{:}\quad & \forall x_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \mathsf{some}\ C_i\langle x_1, \ldots, x_{n_i}\rangle \\
\textsc{Acyclic}\text{:}\quad & \mathsf{no}\ sup_\kappa \cap \mathsf{iden}.
\end{aligned}$$

In the last axiom, $sup_\kappa$ denotes the proper superterm relation for $\kappa$. We will see shortly how to derive it from the selectors.

DISJOINT and EXHAUSTIVE ensure that the discriminators partition $\langle\!\langle \kappa \rangle\!\rangle$. The four remaining axioms, sometimes called the SUGA axioms (after the first letter of each axiom name), ensure that selectors are functions whose domain is given by the corresponding discriminator (SELECTOR), that constructors are total functions (UNIQUE and GENERATOR), and that datatype values cannot be proper superterms of themselves (ACYCLIC). The injectivity of constructors follows from the functionality of selectors.

With this axiomatization, occurrences of $C_i(u_1, \ldots, u_n)$ in FOL are simply mapped to $C_i\langle \mathsf{T}\langle\!\langle u_1 \rangle\!\rangle, \ldots, \mathsf{T}\langle\!\langle u_n \rangle\!\rangle\rangle$, whereas *case t of* $C_1(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_\ell(\bar{x}_\ell) \Rightarrow u_\ell$ is coded as

$$\mathsf{if}\ \mathsf{T}\langle\!\langle t \rangle\!\rangle \subseteq D_1\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1^\star \rangle\!\rangle\ \mathsf{else\ if} \ldots \mathsf{else\ if}\ \mathsf{T}\langle\!\langle t \rangle\!\rangle \subseteq D_\ell\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_\ell^\star \rangle\!\rangle\ \mathsf{else\ none},$$

where $u_i^\star$ denotes the term $u_i$ in which all occurrences of the variables $\bar{x}_i = x_{i1}, \ldots, x_{in_i}$ are replaced with the corresponding selector expressions $S_{i1}(t), \ldots, S_{in_i}(t)$.

Unfortunately, the SUGA axioms admit no finite models if the type $\kappa$ is recursive (and hence infinite), because they force the existence of infinitely many values. The solution is to leave GENERATOR out, yielding SUA. The SUA axioms characterize precisely the subterm-closed finite substructures of an algebraic datatype. In a two-valued logic, this is generally unsound, but Kuncak and Jackson [21] showed that omitting GENERATOR is sound for *existential–bounded-universal* (EBU) sentences—namely, the formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes.

In contrast, in our three-valued setting, omitting GENERATOR is always sound. The construct $C_i\langle r_1, \ldots, r_{n_i}\rangle$ sometimes returns none for non-none arguments, but this is not a problem since our translation of Sect. 4.2 is designed to cope with partiality. Non-EBU formulas such as $True \vee \forall n^{nat}.\, P(n)$ become analyzable when moving to a three-valued logic. This is especially important for complex specifications, because they are likely to contain non-EBU parts that are not needed for finding a model.

*Example 5.1.* The *nat list* instance of *α list* would be axiomatized as follows:

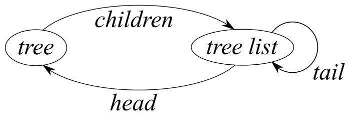
$$\begin{aligned}
\textsc{Disjoint}&: \ \mathsf{no}\ \mathit{nilp} \cap \mathit{consp} \\
\textsc{Exhaustive}&: \ \mathit{nilp} \cup \mathit{consp} \simeq \langle\!\langle \mathit{nat\ list} \rangle\!\rangle \\
\textsc{Selector}_{\mathit{head}}&: \ \forall ys \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle\colon \ \mathsf{if}\ ys \subseteq \mathit{consp}\ \mathsf{then\ one}\ ys.\mathit{head}\ \mathsf{else\ no}\ ys.\mathit{head} \\
\textsc{Selector}_{\mathit{tail}}&: \ \forall ys \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle\colon \ \mathsf{if}\ ys \subseteq \mathit{consp}\ \mathsf{then\ one}\ ys.\mathit{tail}\ \mathsf{else\ no}\ ys.\mathit{tail} \\
\textsc{Unique}_{\mathit{Nil}}&: \ \mathsf{lone}\ \mathit{Nil}\langle\rangle \\
\textsc{Unique}_{\mathit{Cons}}&: \ \forall x \in \langle\!\langle \mathit{nat} \rangle\!\rangle, xs \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle\colon \ \mathsf{lone}\ \mathit{Cons}\langle x, xs\rangle \\
\textsc{Acyclic}&: \ \mathsf{no}\ \mathit{sup}_{\mathit{nat\ list}} \cap \mathsf{iden} \qquad \text{with } \mathit{sup}_{\mathit{nat\ list}} = \mathit{tail}^{+}.
\end{aligned}$$

Examples of subterm-closed list substructures using traditional notation are $\{[], [0], [1]\}$ and $\{[], [1], [2,1], [0,2,1]\}$. In contrast, $L = \{[], [1,1]\}$ is not subterm-closed, because $\mathit{tail}([1,1]) = [1] \notin L$. Given a cardinality, Kodkod systematically enumerates all corresponding subterm-closed list substructures. ■

To generate the proper superterm relation needed for Acyclic, we must consider the general case of mutually recursive datatypes. We start by computing the datatype dependency graph, in which vertices are labeled with datatypes and arcs with selectors. For each selector $S^{\kappa \to \kappa'}$, we add an arc from $\kappa$ to $\kappa'$ labeled $S$. Next, we compute for each datatype a regular expression capturing the nontrivial paths in the graph from the datatype to itself. This can be done using Kleene's construction [18; 19, pp. 51–53]. The proper superterm relation is obtained from the regular expression by replacing concatenation with relational composition, alternative with set union, and repetition with transitive closure.

*Example 5.2.* Let *sym* be an atomic type. The definitions on the left-hand side give rise to the dependency graph on the right-hand side:

**datatype** $\alpha\ \mathit{list} = \mathit{Nil} \mid \mathit{Cons}$ **of** $(\alpha, \alpha\ \mathit{list})$
**datatype** $\mathit{tree} = \mathit{Leaf}$ **of** $\mathit{sym} \mid \mathit{Node}$ **of** $\mathit{tree\ list}$



The selector associated with *Node* is called *children*. The superterm relations are

$$\mathit{sup}_{\mathit{tree}} = (\mathit{children}.\mathit{tail}^{*}.\mathit{head})^{+} \qquad \mathit{sup}_{\mathit{tree\ list}} = (\mathit{tail} \cup \mathit{head}.\mathit{children})^{+}.$$

Notice that in the presence of polymorphism, instances of sequentially declared datatypes can be mutually recursive. ■

With a suitable axiomatization of datatypes as subterm-closed substructures, it is easy to encode **primrec** definitions. A recursive equation $f(C_i(x_1^{\sigma_1}, \ldots, x_m^{\sigma_m}), z_1^{\sigma'_1}, \ldots, z_n^{\sigma'_n}) \simeq t$ is translated to

$$\forall y \in D_i, z_1 \in \langle\!\langle \sigma'_1 \rangle\!\rangle, \ldots, z_n \in \langle\!\langle \sigma'_n \rangle\!\rangle\colon \ \mathsf{T}\langle\!\langle f(y, z_1, \ldots, z_n) \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t^{\star} \rangle\!\rangle,$$

where $t^\star$ is obtained from $t$ by replacing the variables $x_i$ with the selector expressions $S_i(y)$. By quantifying over the constructed values $y$ rather than on the arguments to the constructors, we reduce the number of copies of the quantified body by a factor of $|\langle\!\langle \sigma_1 \rangle\!\rangle| \cdot \ldots \cdot |\langle\!\langle \sigma_n \rangle\!\rangle| \,/\, |\langle\!\langle \kappa \rangle\!\rangle|$ in the SAT problem. Although we focus here on primitive recursion, general well-founded recursion with non-overlapping pattern matching (as defined using, say, Isabelle's **function** package [20]) can be handled in essentially the same way.

*Example 5.3.* The recursive function *cat* from Sect. 3.4 is translated to

$$\forall ys \in nilp,\, zs \in \langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.cat) \simeq zs$$
$$\forall ys \in consp,\, zs \in \langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.cat) \simeq Cons\langle ys.head,\, zs.((ys.tail).cat)\rangle. \quad \blacksquare$$

**Theorem 5.2.** *The encoding of Sect. 5.1 extended with algebraic datatypes and primitive recursion is sound.*

*Proof.* Kuncak and Jackson [21] proved that SUA axioms precisely describe subterm-closed finite substructures of an algebraic datatype, and showed how to generalize this result to mutually recursive datatypes. This means that we can extend the valuation of the descriptors and selectors to obtain a model. For recursion, we can prove $[\![\mathsf{T}\langle\!\langle f(C(x_1,\ldots,x_m),z_1,\ldots,z_n)\rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(C(x_1,\ldots,x_m),z_1,\ldots,z_n)]\!]_M \rfloor$ by structural induction on the value of the first argument to $f$ and extend $f$'s model as in the proof of Theorem 5.1, exploiting the injectivity of constructors. $\square$

## 5.3 Axiomatization of (Co)inductive Predicates

With datatypes and recursion in place, we are ready to consider (co)inductive predicates. Recall from Sect. 3.2 that an inductive predicate is the least fixed point $p$ of the equation $p(\bar{x}) \simeq t[p]$ (where $t[p]$ is some formula involving $p$) and a coinductive predicate is the greatest fixed point. A first intuition would be to take $p(\bar{x}) \simeq t[p]$ as $p$'s definition. In general, this is unsound since it underspecifies $p$, but there are two important cases for which this method is sound.

First, if the recursion in $p(\bar{x}) \simeq t[p]$ is well-founded, the equation admits exactly one solution [13]; we can safely use it as $p$'s specification, and encode it the same way as a recursive function (Sect. 5.2). To ascertain wellfoundedness, we can perform a simple syntactic check to ensure that each recursive call peels off at least one constructor. Alternatively, we can invoke an off-the-shelf termination prover such as AProVE [11] or Isabelle's *lexicographic_order* tactic [6]. Given introduction rules of the form $p(\bar{t}_{i1}) \wedge \cdots \wedge p(\bar{t}_{i\ell_i}) \wedge Q_i \longrightarrow p(\bar{u}_i)$ for $i \in \{1,\ldots,n\}$, the prover attempts to exhibit a well-founded relation $R$ such that $\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\ell_i} Q_i \longrightarrow \langle t_{ij}, u_i\rangle \in R$ holds. This is the approach implemented in Nitpick.

Second, if $p$ is inductive and occurs negatively in the formula, we can replace these occurrences by a fresh constant $q$ satisfying $q(\bar{x}) \simeq t[q]$. The resulting formula is equisatisfiable to the original formula: Since $p$ is a least fixed point, $q$ overapproximates $p$ and thus $\neg\, q(\bar{x}) \Longrightarrow \neg\, p(\bar{x})$. Dually, this method can also handle positive occurrences of coinductive predicates.

To deal with positive occurrences of inductive predicates, we adapt a technique from bounded model checking [3]: We replace these occurrences of $p$ by a fresh predicate $r_k$ defined by the FOL equations

$$r_0(\bar{x}) \simeq \mathit{False} \qquad\qquad r_{\mathit{Suc}\,n}(\bar{x}) \simeq t[r_n],$$

which corresponds to $p$ unrolled $k$ times. In essence, we have made the predicate well-founded by introducing a counter that decreases by one with each recursive call. The above equations are primitive recursive over the datatype *nat* and can be translated using the approach shown in Sect. 5.2. The unrolling comes at a price: The search space for $r_k$ is $k$ times that of $p$ directly encoded as $p(\bar{x}) \simeq t[p]$.

The situation is mirrored for coinductive predicates: Negative occurrences are replaced by the overapproximation $r_k$ defined by

$$r_0(\bar{x}) \simeq \mathit{True} \qquad\qquad r_{\mathit{Suc}\,n}(\bar{x}) \simeq t[r_n].$$

**Theorem 5.3.** *The encoding of Sect. 5.2 extended with (co)inductive predicates is sound.*

*Proof.* We consider only inductive predicates; coinduction is dual. If $p$ is well-founded, the fixed point equation fully characterizes $p$ [13], and the proof is identical to that of primitive recursion in Theorem 5.2 but with recursion induction instead of structural induction. If $p$ is not well-founded, $q \simeq t[q]$ is satisfied by several $q$'s, and by Knaster–Tarski $p \sqsubseteq q$. Substituting $q$ for $p$'s negative occurrences in the FORL formula strengthens it, which is sound. For the positive occurrences, we have $r_0 \sqsubseteq \cdots \sqsubseteq r_k \sqsubseteq p$ by monotonicity of the inductive definition; substituting $r_k$ for $p$'s positive occurrences strengthens the formula. □

Incidentally, we can mobilize FORL's transitive closure to avoid the explicit unrolling for an important class of inductive predicates, *linear inductive predicates*, whose introduction rules are of the form $Q \longrightarrow p(\bar{u})$ (the *base rules*) or $p(\bar{t}) \wedge Q \longrightarrow p(\bar{u})$ (the *step rules*). Informally, the idea is to replace positive occurrences of $p(\bar{x})$ with

$$\exists \bar{x}_0.\; p_{\text{base}}(\bar{x}_0) \wedge p^*_{\text{step}}(\bar{x}_0, \bar{x}),$$

where $p_{\text{base}}(\bar{x}_0)$ iff $p(\bar{x}_0)$ can be deduced from a base rule, $p_{\text{step}}(\bar{x}_0, \bar{x})$ iff $p(\bar{x})$ can be deduced by applying one step rule assuming $p(\bar{x}_0)$, and $p^*_{\text{step}}$ is the reflexive transitive closure of $p_{\text{step}}$. For example, an inductive reachability predicate $\mathit{reach}(s)$ defined inductively would be coded as a set of initial states $\mathit{reach}_{\text{base}}$ and the small-step transition relation $\mathit{reach}_{\text{step}}$. The approach is not so different from explicit unrolling, since Kodkod internally unrolls the transitive closure to saturation. Nonetheless, on some problems the transitive closure approach is several times faster, presumably because Kodkod unfolds the relation inline instead of introducing an explicit counter.

## 5.4 Axiomatization of Coalgebraic Datatypes and Corecursive Functions

Coalgebraic datatypes are similar to algebraic datatypes, but they allow infinite values. For example, the infinite lists $[0, 0, \ldots]$ and $[0, 1, 2, 3, \ldots]$ are possible values of the type *nat llist* of coalgebraic (lazy) lists over natural numbers.

In principle, we could use the same SUA axiomatization for codatatypes as for datatypes (Sect. 5.2). This would exclude all infinite values but nonetheless be sound (although incomplete). However, in practice, infinite values often behave in surprising ways; excluding them would also exclude many interesting models.

It turns out we can modify the SUA axiomatization to support an important class of infinite values, namely those that are $\omega$-regular. For lazy lists, this means lasso-shaped objects such as $[0,0,\ldots]$ and $[8,1,2,1,2,\ldots]$ (where the cycle $1,2$ is repeated infinitely).

The first step is to leave out the ACYCLIC axiom. However, doing only this is unsound, because we might obtain several atoms encoding the same value; for example, $\mathsf{a}_1 = \mathit{LCons}(0, \mathsf{a}_1)$, $\mathsf{a}_2 = \mathit{LCons}(0, \mathsf{a}_3)$, and $\mathsf{a}_3 = \mathit{LCons}(0, \mathsf{a}_2)$ all encode the infinite list $[0,0,\ldots]$. This violates the bisimilarity principle, according to which two values are equal unless they lead to different observations (the observations being $0,0,\ldots$).

For lazy lists, we add the definition

$$\begin{array}{l}\textbf{coinductive} \sim^{(\alpha\ \mathit{llist},\, \alpha\ \mathit{llist}) \to o} \textbf{ where}\\ \mathit{LNil} \sim \mathit{LNil}\\ x \simeq x' \land xs \sim xs' \longrightarrow \mathit{LCons}(x, xs) \sim \mathit{LCons}(x', xs')\end{array}$$

and we require that $\simeq$ coincides with $\sim$ on $\alpha\ \mathit{llist}$ values. More generally, we generate mutual coinductive definitions of $\sim$ for all the codatatypes. For each constructor $C^{(\sigma_1,\ldots,\sigma_n)\to\sigma}$, we add an introduction rule

$$x_1 \approx_1 x_1' \land \cdots \land x_n \approx_n x_n' \longrightarrow C(x_1,\ldots,x_n) \sim C(x_1',\ldots,x_n'),$$

where $\approx_i$ is $\sim^{(\sigma_i,\sigma_i)\to o}$ if $\sigma_i$ is a codatatype and $\simeq$ otherwise. Finally, for each codatatype $\kappa$, we add the axiom

$$\text{BISIMILAR:}\quad \forall y, y' \in \langle\!\langle \kappa \rangle\!\rangle\colon\ y \sim y' \longrightarrow y \simeq y'.$$

With the SUB (SU plus BISIMILAR) axiomatization in place, it is easy to encode **coprimrec** definitions. A corecursive equation $f(y_1^{\sigma_1},\ldots,y_n^{\sigma_1}) \simeq t$ is translated to

$$\forall y_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle,\ldots, y_n \in \langle\!\langle \sigma_n \rangle\!\rangle\colon\ \mathsf{T}\langle\!\langle f(y_1,\ldots,y_n) \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t \rangle\!\rangle.$$

**Theorem 5.4.** *The encoding of Sect. 5.3 extended with coalgebraic datatypes and primitive corecursion is sound.*

*Proof.* Codatatypes correspond to final coalgebras. They are characterized by selectors, which are axiomatized by the SU axioms, and by finality, which is equivalent to the bisimilarity principle [16,26]. Our finite axiomatization gives a subterm-closed substructure of the coalgebraic datatype, which can be extended to yield a FOL model of the complete codatatype, as we did for algebraic datatypes in the proof of Theorem 5.2.

The soundness of the encoding of primitive corecursion is proved by coinduction. Given the equation $f(\bar{y}) \simeq t$, assuming that for each corecursive call $f(\bar{x})$ we have $[\![\mathsf{T}\langle\!\langle f(\bar{x}) \rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(\bar{x})]\!]_M \rfloor$, we must show that $[\![\mathsf{T}\langle\!\langle f(\bar{y}) \rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(\bar{y})]\!]_M \rfloor$. This follows from the soundness of the encoding of the constructs occurring in the right-hand side $t$ and from the hypotheses. □

## 6 Case Study: Lazy Lists

The codatatype $\alpha\ llist$ of lazy lists [26] is generated by the constructors $LNil^{\alpha\ llist}$ and $LCons^{\alpha \to \alpha\ llist \to \alpha\ llist}$. It is of particular interest to (counter)model finding because many basic properties of finite lists do not carry over to infinite lists, often in baffling ways. To illustrate this, we conjecture that appending $ys$ to $xs$ yields $xs$ iff $ys$ is $LNil$:

$$(lcat(xs, ys) \simeq xs) \simeq (ys \simeq LNil).$$

The function $lcat$ is defined corecursively in Sect. 3.4. For the conjecture, our tool Nitpick immediately finds the countermodel $xs = ys = [0,0,\ldots]$, in which a cardinality of 1 is sufficient for $\alpha$ and $\alpha\ llist$, and the bisimilarity predicate $\sim$ is unrolled only once. Indeed, appending $[0,0,\ldots] \neq []$ to $[0,0,\ldots]$ leaves $[0,0,\ldots]$ unchanged.

The next example requires the following lexicographic order predicate:

$$\begin{array}{l}
\textbf{coinductive} \preceq^{(nat\ llist,\, nat\ llist) \to o} \textbf{ where} \\
LNil \preceq xs \\
x \leq y \longrightarrow LCons(x, xs) \preceq LCons(y, ys) \\
xs \preceq ys \longrightarrow LCons(x, xs) \preceq LCons(x, ys)
\end{array}$$

The intention of this definition is to define a linear order on lazy lists of natural numbers, and hence the following properties should hold:

$$\begin{array}{rlrl}
\text{REFL:} & xs \preceq xs & \text{ANTISYM:} & xs \preceq ys \wedge ys \preceq xs \longrightarrow xs \simeq ys \\
\text{LINEAR:} & xs \preceq ys \vee ys \preceq xs & \text{TRANS:} & xs \preceq ys \wedge ys \preceq zs \longrightarrow xs \preceq zs.
\end{array}$$

However, Nitpick finds a counterexample for ANTISYM: $xs = [1,1]$ and $ys = [1]$. On closer inspection, the assumption $x \leq y$ of the second introduction rule for $\preceq$ should have been $x < y$; otherwise, any two lists $xs$, $ys$ with the same head satisfy $xs \preceq ys$. Once we repair the specification, no more counterexamples are found for the four properties up to cardinality 6 for $nat$ and $nat\ llist$, within Nitpick's default time limit of 30 seconds. This is a strong indication that the properties hold. Andreas Lochbihler used Isabelle to prove all four properties [23].

## 7 Related Work

The encoding of algebraic datatypes in FORL has been studied by Kuncak and Jackson [21] and Dunets et al. [10]. Kuncak and Jackson focused on lists and trees. Dunets et al. showed how to handle primitive recursion; their approach to recursion is similar to ours, but the use of a two-valued logic compelled them to generate additional definedness guards. The unrolling of inductive predicates was inspired by bounded model checking [3] and by the Alloy idiom for state transition systems [15, pp. 172–175].

Another inspiration has been Weber's higher-order model finder Refute [31]. It uses a three-valued logic, but sacrifices soundness for precision. Datatypes are approximated by subterm-closed substructures [31, pp. 58–64] that contain *all* datatype values built using up to $k$ nested constructors. This scheme proved disadvantageous in practice, because it generally requires higher cardinalities to obtain the same models as with Kuncak and Jackson's approach. Weber handled (co)inductive predicates by expanding their HOL definition, which in practice does not scale beyond a cardinality of 3 for the predicate's domain because of the higher-order quantifier.

The Nitpick tool, which implements the techniques presented here, is described in a separate paper [5] that covers the handling of higher-order quantification and functions. The paper also presents an evaluation of the tool on various Isabelle/HOL theories, where it competes against Quickcheck [2] and Refute [31], as well as two case studies.

## 8 Conclusion

Despite recent advances in lightweight formal methods, there remains a wide gap between specification languages that lend themselves to automatic analysis and those that are used in actual formalizations. As an example, infinite types are ubiquitous, yet most model finders either spin forever [9, 24], give up immediately [8], or are unsound [1; 28, p. 164; 31] on finitely unsatisfiable formulas.

We identified several commonly used definitional principles and showed how to encode them in first-order relational logic (FORL), the logic supported by the Kodkod model finder and the Alloy Analyzer. Our main contribution has been to develop three ways to translate (co)inductive predicates to FORL, based on wellfoundedness, polarity, and linearity. Other contributions have been to formulate an axiomatization of coalgebraic datatypes that caters for infinite ($\omega$-regular) values and to devise a procedure that computes the acyclicity axiom for mutually recursive datatypes.

Our experience with the counterexample generator Nitpick has shown that the techniques scale to handle real-world specifications, including a security type system and a hotel key card system [5]. Although the tool is fairly new, one user has already reported saving several hours of failed proof attempts thanks to its support for codatatypes and coinductive predicates while developing a formal theory of infinite process traces [22].

***Acknowledgment.*** I want to thank Sascha Böhme, Lukas Bulwahn, Andreas Lochbihler, Tobias Nipkow, Mark Summerfield, and the anonymous reviewers for suggesting many improvements to this paper, and Alexander Krauss for helping to structure it.

## References

1. Ahrendt, W.: Deductive search for errors in free data type specifications using model generation. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 211–225. Springer, Heidelberg (2002)
2. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004, pp. 230–239. IEEE C.S., Los Alamitos (2004)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

4. Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS. Springer, Heidelberg (to appear, 2010)
5. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L. (eds.) ITP-10. LNCS. Springer, Heidelberg (to appear, 2010)
6. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)
7. Church, A.: A formulation of the simple theory of types. J. Symb. Log. 5, 56–68 (1940)
8. Claessen, K., Lillieström, A.: Automated inference of finite unsatisfiability. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS (LNAI), vol. 5663, pp. 388–403. Springer, Heidelberg (2009)
9. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: MODEL (2003)
10. Dunets, A., Schellhorn, G., Reif, W.: Bounded relational analysis of free datatypes. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 99–115. Springer, Heidelberg (2008)
11. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
12. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
13. Harrison, J.: Inductive definitions: Automation and application. In: Schubert, E.T., Windley, P.J., Alves-Foss, J. (eds.) TPHOLs 1995. LNCS, vol. 971, pp. 200–213. Springer, Heidelberg (1995)
14. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
16. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. Bull. EATCS 62, 222–259 (1997)
17. Kleene, S.C.: On notation for ordinal numbers. J. Symb. Log. 3(4), 150–155 (1938)
18. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: McCarthy, J., Shannon, C. (eds.) Automata Studies, pp. 3–42. Princeton University Press, Princeton (1956)
19. Kozen, D.C.: Automata and Computability. Undergrad. Texts in C.S. Springer, Heidelberg (1997)
20. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Auto. Reas. 44(4), 303–336 (2009)
21. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H.C. (ed.) ESEC/FSE 2005 (2005)
22. Lochbihler, A.: Private communication (2009)
23. Lochbihler, A.: Coinduction. In: Klein, G., Nipkow, T., Paulson, L.C. (eds.) The Archive of Formal Proofs (February 2010), http://afp.sourceforge.net/entries/Coinductive.shtml
24. McCune, W.: A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, ANL (1994)
25. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

26. Paulson, L.C.: A fixedpoint approach to implementing (co)inductive definitions. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 148–161. Springer, Heidelberg (1994)
27. Ramananandro, T.: Mondex, an electronic purse: Specification and refinement checks with the Alloy model-finding method. Formal Asp. Comput. 20(1), 21–39 (2008)
28. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer, Heidelberg (2001)
29. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
30. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
31. Weber, T.: SAT-Based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T.U. München (2008)
32. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)

# Combining Theorem Proving and Narrowing for Rewriting-Logic Specifications

Vlad Rusu

Inria Rennes Bretagne Atlantique, France
Vlad.Rusu@inria.fr

**Abstract.** We present an approach for verifying dynamic systems specified in rewriting logic, a formal specification language implemented in the Maude system. Our approach is tailored for invariants, i.e., properties that hold on all states reachable from a given class of initial states. The approach consists in encoding invariance properties into inductive properties written in membership equational logic, a sublogic of rewriting logic also implemented in Maude. The invariants can then be verified using an inductive theorem prover available for membership equational logic, possibly in interaction with narrowing-based symbolic analysis tools for rewriting-logic specifications also available in the Maude environment. We show that it is possible, and useful, to automatically test invariants by symbolic analysis before interactively proving them.

## 1 Introduction

*Rewriting logic* [1], abbreviated as RL in this paper, is a formal specification language, in which a system's dynamics can be expressed by means of *rewrite rules* over a system's *state* defined in some version of equational logic. The adequacy of rewriting logic for specifying dynamic systems has been demonstrated by many practical applications, including programming language semantics [2], security protocols [3], and bioinformatics [4]. There are several systems which implement different variants of this logic, including Maude [5], Elan [6], and CAFEOBJ [7]. *Membership equational logic* [8], hereafter called MEL in this paper, is the logic implemented in Maude as RL's underlying equational logic.

The Maude system [5] consists of a language for expressing RL and MEL specifications, and a set of tools for analysing such specifications and verifying them against user-defined properties. The automatic tools provided by the Maude system include an enumerative, finite state-space searching tool and an enumerative, finite-state model checker for linear temporal logic properties [9] and also for an extension of it called linear temporal logic of rewriting [10]. A symbolic state exploration tool based on narrowing techniques has also recently been made available in Maude [11], and narrowing-based symbolic model checking for linear temporal logic has also been studied [12]. Infinite-state rewriting logic specifications can be verified in Maude with respect to temporal-logic properties, using *equations* to reduce infinite-state spaces to finite-state ones [13].

Our contribution is an approach for verifying invariants of infinite-state RL specifications. Intuitively, an invariant is a predicate that holds in all states that are reachable from a given class of initial states. Our approach consists in encoding the verification of *invariance properties* on the *reachable model* of RL theories, into the verification of MEL properties on the *initial model* of MEL theories. As a consequence, invariance properties can be proved using inductive theorem provers for MEL, such as the ITP tool [14]. Since our formalisation is consistent with that underlying Maude's narrowing-based symbolic analysis tools, our theorem-proving approach can be used in interaction with them.

Specifically, we demonstrate in this paper the usefulness of symbolic simulation in helping the interactive proofs of invariants. Such proofs are performed by induction in the theorem prover, and when the induction step fails, the user must provide the theorem prover with state predicates that (1) are invariants and (2) imply the induction step. While proving (2) is typically automatic - it amounts to proving an implication, the proof of (1) typically has to be performed, again, by induction. Then, assume the user poses a "wrong" state predicate, for which (1) is not provable. Unaware of her error, she will try to prove the invariance of her predicate, also by induction in the theorem prover. The failing induction will lead her to attempt to pose yet other additional auxiliary invariants... in a proof effort that cannot succeed. By contrast, symbolic simulation can automatically falsify invariants by symbolically exploring the system's reachable states up to a given depth, thereby preventing the user from entering dead ends in her proofs.

The rest of the paper is organised as follows. In Section 2 we provide background on MEL and on RL. In Section 3 we recall results about narrowing in the context of RL: the soundness and (under some conditions) the completeness of narrowing for solving reachability problems for rewriting-logic specifications [3]. A class of RL systems satisfying those conditions is identified, and we argue that the class is expressive enough to express many communication protocols. Hence, for systems in that class, narrowing can find all their reachable states starting from a possibly infinite set of initial states, up to a bounded depth; this property is important for our goals - testing invariants before trying to prove them.

In Section 4 we define the notion of a MEL *invariant* $\varphi$ *of a* RL *specification* $\mathcal{R}$ *starting from an initial (possibly, infinite) set of states denoted by a (possibly, non-ground) term* $t_0$, as follows: $\varphi$ is provable in the initial model of the underlying MEL theory $E$ of $\mathcal{R}$, for all states reachable in $\mathcal{R}$ from initial states.

The core of the approach is presented in Section 5. First, we define an automatic translation that takes a RL theory $\mathcal{R}$ and a (possibly, non-ground) term $t_0$, and generates a MEL theory $\mathcal{M}(\mathcal{R}, t_0)$, which enriches the MEL subtheory of $\mathcal{R}$ with a new sort, called *Reachable*, and with the membership axioms that inductively define this sort. We then show (†) if $\mathcal{R}$ is topmost, then, for ground terms $t$, and up to equality *modulo* the equations $E$ of $\mathcal{R}$, the statements "being of sort *Reachable* in $\mathcal{M}(\mathcal{R}, t_0)$" and "being reachable in the reachability model of $\mathcal{R}$ from ground instances of $t_0$" are equivalent. Next, we use the $\mathcal{M}(\mathcal{R}, t_0)$ theory to give an alternative definition of an *invariant* $\varphi$ *of a* RL *theory* $\mathcal{R}$ *starting from a possibly non-ground term* $t_0$, as follows: $\varphi(t)$ is provable in the initial model of the MEL subtheory of $\mathcal{R}$, for all ground terms $t$ that have sort *Reachable* in $\mathcal{M}(\mathcal{R}, t_0)$ (again, up to equality *modulo* $E$). That the two given definitions of invariance are equivalent follows from (†). The advantage of the second definition is that it allows us to prove invariants of RL specifications by induction on the sort *Reachable*, using existing inductive theorem provers for MEL such as Maude's ITP. Section 6 illustrates the theorem-proving of invariants integrated with narrowing-based symbolic falsification of invariants on a Bakery mutual-exclusion algorithm. Section 7 discusses related and future work, and concludes.

## 2 Membership Equational Logic and Rewriting Logic

We briefly present membership equational logic and rewriting logic [1,8,11].

A *membership equational logic* (MEL) *signature* is a tuple $(K, \Sigma, S)$ where $K$ is a set of *kinds*, $\Sigma$ is a $K^* \times K$ indexed family of function symbols $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$, and $S = \{S_k\}_{k\in K}$ is a pairwise disjoint $K$-indexed family of sets of *sorts* - where $S_k$ is the set of sorts of kind $k$. A signature $(K, \Sigma, S)$ is often denoted simply by $\Sigma$; then, $T_\Sigma$ denotes the set of ground terms over signature $\Sigma$. Given a set $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ of *kinded variables*, $T_\Sigma(X)$ denotes the set of terms with free variables in the set $X$. Similarly, $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ denote, respectively, the set of ground terms of kind $k$ and the set of terms of kind $k$ with free variables in the set $X$. A MEL *atomic formula* over $(K, \Sigma, S)$ is either an equality $t = t'$, where $t$ and $t'$ are terms in $T_{\Sigma,k}(X)$, for some kind $k \in K$, or a *membership assertion* $t : s$, where $t$ is a term in $T_{\Sigma,k}(X)$ and $s$ is a sort in $S_k$, for some kind $k \in K$. A MEL *sentence* is a Horn clause

$$(\forall X)t = t' \ \textit{if} \ C, \text{ or} \tag{1}$$
$$(\forall X)t : s \ \textit{if} \ C \tag{2}$$

where the *condition* $C$ has the form $\bigwedge_{i\in I}(u_i = v_i) \wedge \bigwedge_{j\in J}(w_j : s_j)$, for some finite sets of indices $I, J$. Sentences of the form (1) are called *conditional equations*, and sentences of the form (2) are called *conditional memberships*. A sentence is *unconditional* when it does not have a condition.

A MEL *theory* is a tuple $\mathcal{M} = (\Sigma, E)$ that consists of a MEL signature $\Sigma$ and a set of MEL sentences over $\Sigma$. MEL has a complete deduction system [8], in the sense that a formula $\varphi$ is provable from the sentences of a theory $(\Sigma, E)$, denoted as $(\Sigma, E) \vdash \varphi$ (or simply $E \vdash \varphi$), if and only $\varphi$ is semantically valid, i.e., it holds in *all* the *models* of that theory. The standard model of a specification is called its *initial model* [8]. In the initial model, sorts are sets of equivalence classes of ground terms *modulo* the equations $E$, where two ground terms $t, t'$ are in the same equivalence class, denoted by $t =_E t'$, iff $E \vdash (\forall\emptyset)t = t'$. We write $E \vdash_{ind} (\forall X)\varphi$ to say that the sentence $\varphi$ holds in the initial model of $(\Sigma, E)$.

*Example 1.* The specification $NAT$ in Figure 1 defines natural numbers with addition. In the initial model of $NAT$, the natural number $n \geq 1$ is represented by the "sum" $1 + \cdots + 1$ of length $n$, and the term 0 represents the number 0.

$K_{NAT} = \{Nat?\}$ with $S_{Nat?} = \{Nat\}$.
$\Sigma_{NAT(\lambda,Nat?)} = \{0,1\}$.
$\Sigma_{NAT(Nat?Nat?,Nat?)} = \{+\}$.
$\Sigma_{NAT\,w,Nat?} = \emptyset$, otherwise

$$E_{NAT} = \begin{cases} \mathit{0} : Nat, \\ \mathit{1} : Nat, \\ (\forall N)\ 0 + N = N \\ (\forall N, M)\ N + M = M + N \\ (\forall N, M, P)\ (N + M) + P = N + (M + P) \end{cases}$$

**Fig. 1.** A specification of natural numbers

Note the three equations for associativity, communitativity, and unity (ACU). These axioms can either be declared explicitly, or they can be attached to the "+" operator as so-called *equational attributes*. For our purposes we use the former solution with theorem proving, and the latter one with narrowing. The main reason is that Maude's ITP theorem prover that we use does not handle ACU operators. In contrast, *unification* (an essential part of the *narrowing-based symbolic analysis*, which we also use) is *finitary and complete* for ACU operators, provided they are not defined by other equations (or provided the remaining equations have certain technical properties) [15]. The finiteness and completeness of ACU-unification are important for the completeness of narrowing as a symbolic simulation technique for a class of rewriting-logic specifications expressive enough to encode typical communication protocols. We shall come back to this issue at the end of Section 3 after we present rewriting logic and narrowing.

A *rewriting logic* (RL) theory is a tuple $\mathcal{R} = (K, \Sigma, S, E, R)$, - often abbreviated as $(\Sigma, E, R)$ - where $(K, \Sigma, S, E)$ is a MEL theory and $R$ is a set of *rewrite rules* $(\rho)\quad (\forall X)\ l \rightarrow r \quad \mathit{if} \quad C$ where $l, r \in T_{\Sigma,k}(X)$ for some kind $k$ that depends on the rule, and the condition $C$ has the form $\bigwedge_{i \in I}(u_i = v_i) \wedge \bigwedge_{j \in J}(w_j : s_j)$ for some finite sets of indices $I$ and $J$; that is, like for MEL sentences, we consider that only equations and memberships are allowed in the conditions of the rules. Note that in its most general form [16] rewriting logic also allows for *rewrites in conditions* and *frozen arguments*, which we do not consider here. Moreover, we shall only consider *topmost* theories [3]: a theory is *topmost* if there exists a certain kind $k \in K$ such that (i) for all rewrite rules of the above form, $l, r \in T_{\Sigma,k}(X)$, and (ii) no operation in $\Sigma$ takes arguments of the kind $k$. Many authors have shown the adequacy of RL for specifying dynamic systems (including the restricted topmost theories [3] - "almost" all distributed systems can be so described). The idea is to specify the system's state *kind* by equations and membership axioms, and the system's dynamics by (topmost) rewrite rules over the kind of states.

*Example 2.* Mutual exclusion of two processes to a resource can be ensured by the so-called Bakery algorithm. The processes can be in modes *Sleep* (not interested in obtaining the resource), *Try* (when they are trying to obtain the resource) and *Critical* (when they have the resource). To control transitions between these modes, each process has a *ticket*, which is a natural number. The main idea is that a process gets the resource when it has the smallest ticket (because that process has been trying to get the resource for the longest time), or, alternatively, when the other process is not interested in getting the resource.

To specify the system as a RL theory $\mathcal{BAK}$ we use the specification $NAT$ of natural numbers with addition in Figure 1, as well as a (trivial) specification of the kind *Mode?* with the only sort *Mode*, and the three constants $S$, $T$, and $C$ of the sort *Mode*. The states of the system are built using a constructor $\langle \dots \rangle$ that takes two modes and two natural numbers and returns a term in the sort *State*. The evolution of the system is described by the rewrite rules in Figure 2.

Here, $l_1$ and $l_2$ are variables of the sort *Mode* and $t_1, t_2, x$ are variables of the sort *Nat*. We describe the evolution of the first process (the left-hand side column); the evolution of the second process, in the right-hand side column, is similar.

$$\begin{array}{ll}
\langle S, l_2, t_1, t_2\rangle \rightarrow \langle T, l_2, t_2+1, t_2\rangle & \langle l_1, S, t_1, t_2\rangle \rightarrow \langle l_1, T, t_1, t_1+1\rangle \\
\langle T, l_2, t_1, 0\rangle \rightarrow \langle C, l_2, t_1, 0\rangle & \langle l_1, T, 0, t_2\rangle \rightarrow \langle l_1, C, 0, t_2\rangle \\
\langle T, l_2, t_1, t_1+x+1\rangle \rightarrow \langle C, l_2, t_1, t_1+x+1\rangle & \langle l_1, T, t_2+x+1, t_2\rangle \rightarrow \langle l_1, C, t_2+x+1, t_2\rangle \\
\langle C, l_2, t_1, t_2\rangle \rightarrow \langle S, l_2, 0, t_2\rangle & \langle l_1, C, t_1, t_2\rangle \rightarrow \langle l_1, S, t_1, 0\rangle
\end{array}$$

**Fig. 2.** $\mathcal{BAK}$: Bakery algorithm. We only use *unconditional* rewrite rules, since conditional narrowing is outside the scope of Maude's current implementation of narrowing.

The first rule moves the process from the *Sleep* to the *Try* mode, and changes the value of its ticket $t_1$, by "assigning" to it the term $t_2 + 1$. Then, the first process may move to the *Critical* mode if (a) the other process has its ticket equal to 0, or (b) the first process has the smallest ticket. The latter condition is obtained by having the ticket of the second process denoted by the term $t_1+x+1$, for some $x$ of the sort $Nat$, where $t_1$ is the ticket of the first process. Finally, the first process goes back to the *Sleep* mode and sets its ticket back to zero.

Assume that we want to simulate the behaviours of the $\mathcal{BAK}$ starting from a possibly infinite class of initial states. Let the initial states be denoted by the term $\langle S, S, t, t\rangle$, in which both processes are in the *Sleep* modes and have the same initial value $t$ for their tickets, where $t$ is a variable of the sort *Nat* - the actual initial value of the tickets is left unspecified. The desired simulation cannot be performed using Maude's enumerative state-exploration tools, because those tools require a unique initial state, i.e., a ground term. By contrast, narrowing can simulate the executions of our system, starting from a non-ground term.

*Reachability.* The notation $\mathcal{R} \vdash (\forall X)t_0 \rightarrow t$ expresses the fact that the term $t \in T_\Sigma(X)$ is *provable* from the term $t_0 \in T_\Sigma(X)$ in the deduction system of $\mathcal{R}$ (which amounts to applying the rewrite rules of $\mathcal{R}$ *modulo* the equations $E$ of $\mathcal{R}$). The *reachability model* [16] of $\mathcal{R}$ is a *transition system* whose states are equivalence classes of ground terms *modulo* $E$. For all states $[t]_E, [t']_E$, there is a transition $[t]_E \rightarrow_{\mathcal{R}} [t']_E$ in this model iff there exists a proof $\mathcal{R} \vdash (\forall \emptyset)t \rightarrow t'$ using exactly one rewrite rule of $\mathcal{R}$. We denote by $[t]_E \rightarrow^*_{\mathcal{R}} [t']_E$ the fact that the state $[t']_E$ is reachable from the state $[t]_E$ in the reachability model of $\mathcal{R}$.

## 3 Narrowing, and a Class of Systems It Can Analyse

In the context of rewriting logic, narrowing is used for symbolically solving *reachability problems* [3,11] (and more generally, for symbolically model checking linear temporal-logic properties [12]). We consider reachability problems of the form *"given terms $t_0 \in T_\Sigma(X)$ and $t \in T_\Sigma$ does there exist a (ground) substitution $\sigma : X \to T_\Sigma$ such that $[t_0\sigma]_E \to^*_{\mathcal{R}} [t]_E$ holds"* for topmost theories $\mathcal{R}$, whose rewrite rules are *unconditional* and do not have supplementary variables in their right-hand sides wth respect to their left-hand sides - to simplify matters and to be consistent with the current implementation of narrowing in Maude.

Given a substitution $\sigma : X \mapsto T_\Sigma(X)$, we write $t_1 \overset{\sigma}{\leadsto}_{\mathcal{R}} t_2$ if there exists in $\mathcal{R}$ a rule $(\rho) \quad (\forall X)\ l \to r$ such that the variables occuring in $t_1$ and $l$ are disjoint and such that $E \vdash t_1\sigma = l\sigma$ and $E \vdash r\sigma = t_2$. That is, it is provable in the MEL subtheory $(\Sigma, E)$ of $\mathcal{R}$ that $\sigma$ is a *unifier* for $t_1$ and the left-hand side $l$ of the rule $(\rho)$ and that $\sigma$ *matches* the right-hand side $r$ of the rule with the term $t_2$.

For $t_1, t'_1, t_2, t'_2 \in T_\Sigma(X)$ we write $t'_1 \overset{\sigma}{\leadsto}_{\mathcal{R},E} t'_2$ if $t'_1 =_E t_1 \overset{\sigma}{\leadsto}_{\mathcal{R}} t_2 =_E t'_2$. The narrowing relation $\leadsto_{\mathcal{R},E} \subseteq T_\Sigma(X) \times T_\Sigma(X)$ is defined by $t_1 \leadsto_{\mathcal{R},E} t_2$ iff $t_1 \overset{\sigma}{\leadsto}_{\mathcal{R},E} t_2$ for some substitution $\sigma$. Let $\leadsto^*_{\mathcal{R},E}$ be the reflexive transitive closure of $\leadsto_{\mathcal{R},E}$. The *soundness of narrowing for solving reachability problems* in Maude [11] says essentially that for all terms $t_0 \in T_\Sigma(X)$ and $t \in T_\Sigma$, if $t_0 \leadsto^*_{\mathcal{R},E} t$ then there exists a ground substitution $\sigma : X \mapsto T_\Sigma$ such that $[t_0\sigma]_E \to^*_{\mathcal{R}} [t]_E$.

On the other hand, *completeness* of narrowing, meaning that narrowing $t_0 \leadsto^*_{\mathcal{R},E} t$ "finds" in some sense all solutions $\sigma$ such that $[t_0\sigma]_E \to^*_{\mathcal{R}} [t]_E$, does not hold in general, but only under some technical conditions [3]. The most important condition is that unification modulo $E$ be *finitary and complete*; that is, for any equation of the form $t_1 =_E t_2$, for $t_1, t_2 \in T_\Sigma(X)$, the algorithm returns a finite set of substitutions, which are all the solutions of the equation[1].

And this is precisely the case when $E$ consists of ACU axioms for some operations in $\Sigma$, such as those defined for the "+" operation in our specification of natural numbers (Figure 1), or the encoding of sets in MEL based on an ACU "union" operation. These observations are important because they suggest a class of systems that can be effectively symbolically simulated by narrowing, in the sense that narrowing eventually "reaches" all reachable states. The Bakery algorithm in Figure 2 is one such system, thanks to the ACU-based definition of natural numbers given in Figure 1. More generally, finite control and possibly unbounded counters, typically encountered in such protocols, can be encoded using our ACU-based encoding of natural numbers. Even more generally, *communication protocols* with unordered channels also fall in this class - by encoding unordered channels using sets constructed with an ACU definition of union.

One limitation remaining is that such communication protocols often require conditions on the counters: such as, e.g., the conditions on the *tickets* in the

[1] The other conditions are (in addition to those posed at the beginning of this section) that the theory $(\Sigma, E)$ is in the *order-sorted* fragment of MEL and that that the equations be *regular and sort-preserving*: left-hand and right-hand sides have the same variables, and left-hand side does not have a greater sort than right-hand side.

Bakery algorithm. However, we can encode such *affine* conditions - comparisons of linear-arithmetic terms with constants - by unconditional rules, as follows. Assume a system endoded in RL having one conditional topmost rule of the form $\langle l\rangle \to \langle r\rangle$ *if* $\Sigma_{i=1}^{n} a_i x_i > b$ (and possibly other rules). We enrich the sort *State* with two integer components. The initial states will now have the form $\langle I, \Sigma_{i=1}^{n} a_i x_i, b\rangle$, where $I$ is the expression denoting initial the states of the original system. Each rule except the one we are encoding, of the form $\langle l'\rangle \Rightarrow \langle r'\rangle$ *if* $C'$, becomes $\langle l', x, y\rangle \Rightarrow \langle r', x, y\rangle$ *if* $C'$ - that is, all the rules except the one we are encoding leaves the "new" components of the *State* unchanged. Our rule of interest becomes $\langle l, x, x+n+1\rangle \Rightarrow \langle r, x, x+n+1\rangle$, where the effect of checking the condition to "see" if the rule can be applied on a term is achieved by *unifying* the left-hand side of our rule: $\langle l, x, x+n+1\rangle$ with that term. By generalising this encoding to several linear-arithmetic conditions and to several rules, we obtain an unconditional system equivalent to a conditional one.

Thus, narrowing can effectively simulate a class of systems expressive enough to encode communication protocols with finite control, counters, and channels.

## 4 Invariants of Rewrite Theories

We continue by discussing in this section the notion of *invariant* for a rewrite theory. In general, a predicate over the states of a dynamic system is an invariant if the predicate holds in all the states of the system that are reachable from a given class of initial states. To formalize this notion in the case of a system specified in a (topmost) RL theory $\mathcal{R}$, we have to answer the following questions:

1. how are the *states* and the *dynamics* to be specified?
2. how are the state *predicates* to be formalized?
3. when are the predicates to be considered as *holding* in a state?

For item (1) we adopt the usual RL representations: states are equivalence classes (*modulo* the equations $E$ of $\mathcal{R}$) of ground terms of a certain kind $[State]$. There is a possibly infinite set of initial states denoted by a term $t_0$, possibly with variables, of the kind $[State]$; then, initial states are equivalence classes of ground instances of $t_0$. Regarding the dynamics of the system, it shall naturally be defined by reachability in the reachability model of $\mathcal{R}$.

With regards to item (2): state predicates shall be formalised by Horn sentences of the form $(\forall x : [State])(\forall Y)\varphi$ having a free variable $x$ of the kind $[State]$ (and possibly other free variables in the set $Y$, with $x \notin Y$). Finally, for item (3), a state predicate $\varphi$ shall be considered to hold in a state $t$ when the predicate $\varphi(t/x)$, obtained from $\varphi$ by substituting the variable $x$ with the term $t$, holds in the initial model of the MEL subtheory of the RL theory: $E \vdash_{ind} (\forall Y)\varphi(t/x)$.

In summary, when a system is specified as a topmost RL theory $\mathcal{R}$, we formalize the intuitive notion of an invariant as a state predicate $\varphi$ holding in all states are reachable from an initial state - denoted by $\langle \mathcal{R}, t_0\rangle \vdash_{ind} \Box\varphi$ - as follows.

**Definition 1.** $\langle \mathcal{R}, t_0\rangle \vdash_{ind} \Box\varphi$ *if for all ground terms* $t \in T_\Sigma$, *and all ground substitutions* $\sigma : X \mapsto T_\Sigma$, $[t_0\sigma]_E \to_{\mathcal{R}}^{*} [t]_E$ *implies* $E \vdash_{ind} (\forall Y)\varphi(t/x)$.

*Example 3.* Consider the RL specification $\mathcal{BAK}$ from Example 2. We have seen that the states of the system are quadruples consisting of two modes and two natural numbers built using the constructor $\langle \dots \rangle$ of the sort *State*. The mutual exclusion between readers and writers is encoded as the state predicate *mutex*:

$$(\forall\, x : [State])(\forall\, t_1, t_2 : Nat).\ mutex(\langle C, C, t_1, t_2\rangle) = false$$
$$(\forall\, x : [State])(\forall\, t_1, t_2 : Nat, l_1, l_2 : Mode).\ l_1 \neq C \Rightarrow mutex(\langle l_1, l_2, t_1, t_2\rangle) = true$$
$$(\forall\, x : [State])(\forall\, t_1, t_2 : Nat, l_1, l_2 : Mode).\ l_2 \neq C \Rightarrow mutex(\langle l_1, l_2, t_1, t_2\rangle) = true$$

The invariance of *mutex* on the $\mathcal{BAK}$ system starting from all states denoted by the term $\langle S, S, t, t\rangle$ with the variable $t : Nat$, is written $\langle \mathcal{BAK}, \langle S, S, t, t\rangle\rangle \vdash_{ind} \Box mutex$ and defined by: for all ground terms $t$ of kind $[State]$, and for each ground term $n : Nat$, $\langle S, S, n, n\rangle \rightarrow^*_{\mathcal{BAK}} [t]_{E_{\mathcal{BAK}}}$ implies $NAT \vdash_{ind} mutex(t/x)$.

Before we proceed to verifying invariants by theorem proving, we show how invariants can be disproved using Maude's narrowing-based symbolic analysis.

*Disproving invariants.* Disproving an invariance statement $\langle \mathcal{R}, t_0\rangle \vdash_{ind} \Box\varphi$ amounts to finding a ground substitution $\sigma$ and a sequence $[t_0\sigma]_E \rightarrow^*_{\mathcal{R}} [t]_E$ such that $E \not\vdash_{ind} \varphi(t/x)$. If we find a narrowing sequence $t_0 \rightsquigarrow^*_{\mathcal{R},E} t$ such that $E \not\vdash_{ind} \varphi(t/x)$, then, by soundness of narrowing there exists a sequence $[t_0\sigma]_E \rightarrow^*_{\mathcal{R}} [t]_E$ such that $E \not\vdash_{ind} \varphi(t/x)$, hence, $\langle \mathcal{R}, t_0\rangle \vdash_{ind} \Box\varphi$ is disproved.

The completeness of narrowing says that all such "disproofs" will eventually be found by narrowing. Concretely, such sequences $t_0 \rightsquigarrow^*_{\mathcal{R},E} t$ can be found by Maude's `search` command. Consider a variant $\mathcal{BAK}'$ of our running example, in which the term 1 is replaced everywhere by the term 0. The invariance statement $\langle \mathcal{BAK}', \langle S, S, t, t\rangle\rangle \vdash_{ind} \Box mutex$ can be disproved (falsified) by Maude's following command: `search`$\langle S, S, t, t\rangle \rightsquigarrow^* \langle C, C, t_1, t_2\rangle$, which immediately finds the term $\langle C, C, 0, 0\rangle$ violating the *mutex* predicate. A more involved use of the `search` command to check *auxiliary* invariants that are needed in an inductive proof of the "main" mutual-exclusion invariant of the $\mathcal{BAK}$ system is shown in Section 6.

## 5 Theorem Proving for Invariance Properties

The previous section discussed the falsification of invariants. In this section we propose an approach for *proving* invariants. The structure of the section is as follows. We first define an automatic translation that takes a topmost RL theory $\mathcal{R}$ and a term $t_0$, possibly with variables, and generates a MEL theory $\mathcal{M}(\mathcal{R}, t_0)$, which enriches $\mathcal{R}$ with a sort called *Reachable* and with memberships defining this sort. We show that for all ground terms $t$, the state $[t]_E$ is reachable in the initial model of $\mathcal{R}$ from an initial state $[t_0\sigma]$ if and only if the term $t$ has the sort *Reachable* in the initial model of $\mathcal{M}(\mathcal{R}, t_0\sigma)$. Then, we prove that, for any state predicate $(\forall x : [State])(\forall Y)\varphi$, the statements $\langle \mathcal{R}, t_0\rangle \vdash_{ind} \Box\varphi$ and $\mathcal{M}(\mathcal{R}, t_0) \vdash_{ind} (\forall x : [State])(\forall Y)(x : Reachable \Rightarrow \varphi)$ are equivalent. This equivalence is the basis for proving invariants using inductive theorem provers, such as the ITP tool.

In the following definition, we "encode" reachability in a RL theory $\mathcal{R}$ (starting from a possibly non-ground term $t_0$) in a MEL theory $\mathcal{M}(\mathcal{R}, t_0)$ using a membership axiom for $t_0$ and a membership axiom $\mu(\rho)$ for each rule $\rho$ in $\mathcal{R}$.

**Definition 2.** *Consider a* RL *theory* $\mathcal{R} = (K, \Sigma, S, E, R)$*, a sort* $State \in S$*, and a term* $t_0 \in T_{\Sigma,[State]}(X)$*. We denote by* $\mathcal{M}(\mathcal{R}, t_0)$ *the following* MEL *theory* $(K_{\mathcal{M}(\mathcal{R},t_0)}, \Sigma_{\mathcal{M}(\mathcal{R},t_0)}, S_{\mathcal{M}(\mathcal{R},t_0)}, E_{\mathcal{M}(\mathcal{R},t_0)})$ *constructed as follows:*

- $K_{\mathcal{M}(\mathcal{R},t_0)} = K$
- $\Sigma_{\mathcal{M}(\mathcal{R},t_0)} = \Sigma$
- $S_{\mathcal{M}(\mathcal{R},t_0)} = S \cup S'_{[State]}$ *with* $S'_{[State]} = S_{[State]} \cup \{Reachable\}$ *and* $Reachable \notin S$
- $E_{\mathcal{M}(\mathcal{R},t_0)} = E \cup \{(\forall X) t_0 : Reachable\} \cup \{\mu(\rho) | \rho \in R\}$*, with* $\mu((\forall X)\ l \to r\ \mathit{if}\ \ C))$ *being the membership* $(\forall X)\ r : Reachable\ \mathit{if}\ \ l : Reachable \wedge C$*.* □

*Example 4.* Consider the Bakery system given in Example 2. The MEL theory $\mathcal{M}(\mathcal{BAK}, \langle S, S, t, t\rangle)$ consists of the following elements:

- $K_{\mathcal{M}(\mathcal{BAK},\langle S,S,t,t\rangle)} = K_{\mathcal{BAK}}$.
- $\Sigma_{\mathcal{M}(\mathcal{BAK},\langle S,S,t,t\rangle)} = \Sigma_{\mathcal{BAK}}$.
- $S_{\mathcal{M}(\mathcal{BAK},\langle S,S,t,t\rangle)} = S_{\mathcal{BAK}} \cup S'_{[State]}$, with $S'_{[State]} = S_{\mathcal{BAK}_{[State]}} \cup \{Reachable\}$
- $E_{\mathcal{M}(\mathcal{BAK},\langle S,S,t,t\rangle)} = E_{\mathcal{BAK}} \cup E'$, where $E'$ is the set of memberships axioms shown in Figure 3.

$\langle S, S, t, t\rangle : Reachable$

$\langle T, l_2, t_2 + 1, t_2\rangle : Reachable\ \mathit{if}\ \ \langle S, l_2, t_1, t_2\rangle : Reachable$
$\langle C, l_2, t_1, 0\rangle : Reachable\ \mathit{if}\ \ \langle T, l_2, t_1, 0\rangle : Reachable$
$\langle C, l_2, t_1, t_1{+}x{+}1\rangle : Reachable\ \mathit{if}\ \ \langle T, l_2, t_1, t_1{+}x{+}1\rangle : Reachable$
$\langle S, l_2, 0, t_2\rangle : Reachable\ \mathit{if}\ \ \langle C, l_2, t_1, t_2\rangle : Reachable$

$\langle l_1, T, t_1, t_1 + 1\rangle : Reachable\ \mathit{if}\ \ \langle l_1, S, t_1, t_2\rangle : Reachable$
$\langle l_1, C, 0, t_2\rangle : Reachable\ \mathit{if}\ \ \langle l_1, T, 0, t_2\rangle : Reachable$
$\langle l_1, C, t_2{+}x{+}1, t_2\rangle : Reachable\ \mathit{if}\ \ \langle l_1, T, t_2{+}x{+}1, t_2\rangle : Reachable$
$\langle l_1, S, t_1, 0\rangle : Reachable\ \mathit{if}\ \ \langle l_1, C, t_1, t_2\rangle : Reachable$

**Fig. 3.** Membership axioms for $\mathcal{M}(\mathcal{BAK}, \langle S, S, t, t\rangle)$

**Lemma 1.** *Consider a* RL *theory* $\mathcal{R} = (K, \Sigma, S, E, R)$*, with* $State \in S$ *and* $t \in T_{\Sigma,[State]}$*. For all* $t' \in T_{\Sigma,[State]}$*,* $[t]_E \to^*_{\mathcal{R}} [t']_E$ *iff* $\mathcal{M}(\mathcal{R}, t) \vdash t' : Reachable$*.*

*Proof.* The idea of the proof is that each transition in the reachability model of $\mathcal{R}$, generated by using a rule $(\rho)$ of $\mathcal{R}$, can be emulated by an deduction in the proof system of $\mathcal{M}(\mathcal{R}, t)$, using the membership $\mu(\rho)$ given in Definition 2.

($\Rightarrow$) By induction on the length of the sequence $[t]_E \to^*_{\mathcal{R}} [t']_E$.

If the length is 0 then $E \vdash (\forall\emptyset)t = t'$. The membership $t : Reachable$ in $\mathcal{M}(\mathcal{R}, t)$ implies $\mathcal{M}(\mathcal{R}, t) \vdash t : Reachable$, hence, $\mathcal{M}(\mathcal{R}, t) \vdash t' : Reachable$.

Assume the statement holds for sequence of length $n$. Any sequence $[t]_E \to^{n+1}_{\mathcal{R}} [t']_E$ of length $n + 1$ can be decomposed into $[t]_E \to^n_{\mathcal{R}} [t'']_E \to_{\mathcal{R}} [t']_E$, such that the last step uses a rule $(\rho)$ $(\forall X)l \to r\ \mathit{if}\ \ C$ with a ground substitution $\sigma$.

1. then, $t'' \equiv t''\sigma =_E l\sigma$ because $t''$ is ground (and $\equiv$ denotes syntactical equality), $t' =_E r\sigma$, and $E \vdash C\sigma$. The latter implies *a fortiori* $\mathcal{M}(\mathcal{R}, t) \vdash C\sigma$;

2. by induction, $\mathcal{M}(\mathcal{R},t) \vdash t'' : \mathit{Reachable}$, hence, $\mathcal{M}(\mathcal{R},t) \vdash l\sigma : \mathit{Reachable}$;
3. hence, using the membership $(\mu(\rho))\ r : \mathit{Reachable\ if\ \ } l : \mathit{Reachable} \wedge C$ from Definition 2 with $\sigma$, $\mathcal{M}(\mathcal{R},t) \vdash r\sigma : \mathit{Reachable}$, i.e., $\mathcal{M}(\mathcal{R},t) \vdash t' : \mathit{Reachable}$.

$(\Leftarrow)$ By induction on the length of the proof $\mathcal{M}(\mathcal{R},t) \vdash t' : \mathit{Reachable}$, where by length we here mean the number of applications of memberships of the form $(\mu(\rho))\ r : \mathit{Reachable\ if\ \ } l : \mathit{Reachable}{\wedge}C$ generated from rules $(\rho)\ (\forall X)l \to r\ \mathit{if}\ \ C$.

If the length is 0 then $t' : \mathit{Reachable}$ has been proved using the membership $t : \mathit{Reachable}$ for the initial term, hence, $t =_E t'$ and $[t]_E \to^*_{\mathcal{R}} [t']_E$ follows.

Assume the statement holds for sequence of length $n$. Any proof $\mathcal{M}(\mathcal{R},t) \vdash t' : \mathit{Reachable}$ of length $n+1$ can be decomposed into a proof $\mathcal{M}(\mathcal{R},t) \vdash t'' : \mathit{Reachable}$ of length $n$, for some $t'' \in T_\Sigma$, followed by an application of a membership $(\mu(\rho))\ r : \mathit{Reachable\ if\ \ } l : \mathit{Reachable} \wedge C$ with a ground substitution $\sigma$ such that $t' =_E r\sigma$, $t'' =_E l\sigma$, and $\mathcal{M}(\mathcal{R},t) \vdash C\sigma$. Since the sort *Reachable* is "new" in $\mathcal{M}(\mathcal{R},t)$, it does not occur in the condition $C$, hence, $E \vdash C\sigma$. Since $t''$ is ground, $t'' \equiv t''\sigma =_E l\sigma$. Hence, the rule $(\rho)\ (\forall X)l \to r\ \mathit{if}\ \ C$ can be applied on $t''$ and generates the transition $[t'']_E {\to_{\mathcal{R}}} [t']_E$. By induction hypothesis, $[t]_E \to^*_{\mathcal{R}} [t'']_E$. The transitivity of the $\to^*_{\mathcal{R}}$ relation concludes. □

We have proved the equivalence between $[t]_E \to^*_{\mathcal{R}} [t']_E$ and $\mathcal{M}(\mathcal{R},t) \vdash t' : \mathit{Reachable}$ for ground terms $t, t'$. In particular, in our setting where the terms $t$ are ground instances of the (possibly, non-ground) term $t_0$, we obtain the equivalence between *for all ground substitutions* $\sigma$, $[t_0\sigma] \to^*_{\mathcal{R}} [t]_E$ and *for all ground substitutions* $\sigma$, $\mathcal{M}(\mathcal{R},t_0\sigma) \vdash t : \mathit{Reachable}$. However, in order to reason by induction on *Reachable*, we need a different hypothesis, namely, $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} t : \mathit{Reachable}$. The following lemma bridges the gap between those statements.

**Lemma 2.** *For* $t_0 \in T_\Sigma(X)$ *and* $t \in T_\Sigma$, $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} t : \mathit{Reachable}$ *if and only if* $\mathcal{M}(\mathcal{R},t_0\sigma) \vdash t : \mathit{Reachable}$ *for all ground substitutions* $\sigma : X \mapsto T_\Sigma$.

*Proof.* Let us denote by $\mathcal{M}(\mathcal{R})$ the MEL theory obtained by removing the membership $(\forall X)t_0 : \mathit{Reachable}$ from the theory $\mathcal{M}(\mathcal{R},t_0)$ in Definition 2. Then, $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} t : \mathit{Reachable}$ iff $\mathcal{M}(\mathcal{R}) \vdash_{\mathit{ind}} ((\forall X)t_0 : \mathit{Reachable} \Rightarrow t : \mathit{Reachable})$. Since truth in the initial model for a statement is equivalent to deduction of all ground instances of that statement, we obtain that the last entailment is equivalent to $\mathcal{M}(\mathcal{R}) \vdash (t_0\sigma : \mathit{Reachable} \Rightarrow t : \mathit{Reachable})$ for all ground substitutions $\sigma$, itself equivalent to $\mathcal{M}(\mathcal{R},t_0\sigma) \vdash t : \mathit{Reachable}$ for all ground substitutions $\sigma$. □

**Theorem 1.** *Consider a* RL *theory* $\mathcal{R} = (K,\Sigma,S,E,R)$, *with* $\mathit{State} \in S$, *a term* $t_0 \in T_{\Sigma,[\mathit{State}]}(X)$, *and a state predicate* $(\forall x : [\mathit{State}], \forall Y)\varphi$. *Then* $\langle\mathcal{R},t_0\rangle \vdash_{\mathit{ind}} \Box\varphi$ *if and only if* $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} (\forall x : [\mathit{State}])(\forall Y)(x : \mathit{Reachable} \Rightarrow \varphi)$.

*Proof.* Since $x \notin Y$, $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} (\forall x : [\mathit{State}])(\forall Y)(x : \mathit{Reachable} \Rightarrow \varphi)$ is equivalent to $\mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} (\forall x : [\mathit{State}])(x : \mathit{Reachable} \Rightarrow (\forall Y)\varphi)$, and using the fact that truth in the initial model for a statement is equivalent to deduction of ground instances of that statement, we obtain equivalently $\forall t \in T_{\Sigma,[\mathit{State}]}.\, \mathcal{M}(\mathcal{R},t_0) \vdash_{\mathit{ind}} (t : \mathit{Reachable} \Rightarrow (\forall Y)\varphi(t/x))$. Then, using the equivalence $A \vdash (B \Rightarrow C)$ iff ($A \vdash B$ implies $A \vdash C$) we obtain equivalently

$$\forall t \in T_{\Sigma,[State]}.\ \ ([\mathcal{M}(\mathcal{R}, t_0) \vdash_{ind} t : \mathit{Reachable}] \text{ implies } [\mathcal{M}(\mathcal{R}, t_0) \vdash_{ind} (\forall Y)\varphi(t/x)]) \tag{3}$$

By using Lemmas 1 and 2, the left-hand side of the above implication is equivalent to *for all ground substitutions* $\sigma$, $[t_0\sigma]_E \rightarrow^*_{\mathcal{R}} [t]_E$. Then, the implication (3) is equivalent to (‡) *for all* $t \in T_{\Sigma,[State]}$ *and all ground substitutions* $\sigma$, $[t_0\sigma]_E \rightarrow^*_{\mathcal{R}} [t]_E$ *implies* $\mathcal{M}(\mathcal{R}, t_0) \vdash_{ind} (\forall Y)\varphi(t/x)$. Finally, we note that truth in the initial model of $\mathcal{M}(\mathcal{R}, t_0)$ and truth in the initial model of $E$ are equivalent, for all statements that do not refer to the "new" sort *Reachable*, because, for, the truth of such statements, the memberships defining *Reachable* in $\mathcal{M}(\mathcal{R}, t_0)$ are irrelevant. And $\varphi$ does not refer to this sort, precisely because it is "new". Hence, the last statement (‡) in our chain of equivalences is itself equivalent to *for all* $t \in T_{\Sigma,[State]}$ *and all ground substitutions* $\sigma$, $[t_0\sigma]_E \rightarrow^*_{\mathcal{R}} [t]_E$ *implies* $E \vdash_{ind} (\forall Y)\varphi(t/x)$, which by Definition 1 is $\langle \mathcal{R}, t_0 \rangle \vdash_{ind} \Box\varphi$. □

## 6 Testing Invariants before Proving Them

The results in the previous section show that proving invariants is equivalent to proving inductive theorems in the initial model of a MEL theory, thus, invariants can be proved by induction. Consider the specification $\mathcal{BAK}$ and its *mutex* predicate. To prove the statement $\langle \mathcal{BAK}, \langle S, S, t, t \rangle \rangle \vdash_{ind} \Box \mathit{mutex}$, we prove $(\forall\, x)(x : \mathit{Reachable} \Longrightarrow \mathit{mutex}(x) = \mathit{true})$ in the initial model of $\mathcal{M}(\mathcal{BAK}, \langle S, S, t, t \rangle)$ using the ITP tool. We describe that proof in some detail, and show that narrowing-based symbolic simulation is really useful in preventing the user from taking a wrong direction in the proof.

The proof goes by induction on the sort *Reachable*. This generates nine subgoals: one for the membership defining the initial state, and eight for the eight other memberships defining the sort *Reachable* (all memberships shown in Fig. 3).

The subgoal for the initial states is automatically proved by the ITP. Out of the eight remaining subgoals, four are also automatically proved by the ITP. Those are the subgoals corresponding to the memberships whose left-hand sides are states where at least one process is *not* in the *Critical* mode. These are, for the first process: $\langle T, l_2, t_2{+}1, t_2 \rangle : \mathit{Reachable}\ \mathit{if}\ \ \langle S, l_2, t_1, t_2 \rangle : \mathit{Reachable}$ and $\langle S, l_2, 0, t_2 \rangle : \mathit{Reachable}\ \mathit{if}\ \ \langle C, l_2, t_1, t_2 \rangle : \mathit{Reachable}$, and the symmetrical ones for the second process. In the left-hand sides of these memberships, the *mutex* predicate obviously holds, and the ITP tool "realises" this.

The remaining subgoals cannot be automatically proved by the ITP, because it needs additional information that only the user can provide.

Corresponding to the following membership of the first process:

$$\langle C, l_2, t_1, t_1 + x + 1 \rangle : \mathit{Reachable}\ \mathit{if}\ \ \langle T, l_2, t_1, t_1 + x + 1 \rangle : \mathit{Reachable}$$

the ITP presents us with essentially the following subgoal, written as a *sequent*:

$$\frac{\begin{array}{c}\langle T, l_2, t_1, t_1+x+1\rangle{:}\mathit{Reachable}\\ \mathit{mutex}(\langle T, l_2, t_1, t_1+x+1\rangle)\end{array}}{\mathit{mutex}(\langle C, l_2, t_1, t_1+x+1\rangle)}$$

That is, using the hypotheses "above" the line, one has to prove the conclusion "below" the line. In order to simplify her proof, the user performs a case splitting on the variable $l_2$, which generates three sub-subgoals for the given subgoal. Two of them are automatically proved by the ITP, because their conlusions are $\mathit{mutex}(C, T, t_1, t_1+x+1\rangle)$ and $\mathit{mutex}(C, S, t_1, t_1+x+1\rangle)$, which obviously hold. However, the third sub-subgoal is not proved by the ITP: it has the form

$$\frac{\begin{array}{c}\langle T, C, t_1, t_1+x+1\rangle{:}\mathit{Reachable}\\ \mathit{mutex}(\langle T, C, t_1, t_1+x+1\rangle)\end{array}}{\mathit{mutex}(\langle C, C, t_1, t_1+x+1\rangle)} \tag{4}$$

By examining the hypotheses in the subgoal (4), the user realises that the second one is trivially *true*, hence, it is useless; and that the conclusion is trivially *false*. The only remaining possibility for proving the subgoal is therefore to prove that the first hypothesis: $\langle T, C, t_1, t_1+x+1\rangle : \mathit{Reachable}$ does not hold. After some thinking, the user realises that indeed, states of the form $\langle T, C, t_1, t_1+x+1\rangle$ should not be reachable, because the very basic principle of the Bakery algorithm is that the process that is in the critical section should have the *smallest ticket*; but that is precisely *not* the case in the states of the above form.

Happy with her reasoning, the user poses the following lemma to the ITP:

$$\frac{\langle l_1, l_2, t_1, t_1+x\rangle : \mathit{Reachable}}{l_2 \neq C} \tag{5}$$

She postpones proving (5), and confidently uses it to sucessfully prove the subgoal (4). Eventually, she completes the proof of the main invariant *mutex*, and returns to proving (5). However, no matter how hard she tries, she does not succeed... of course, because the lemma is not true! Indeed, had the user tried to falsify (5) using Maude's narrowing-based `search` command, she would have realised her error: `search` $\langle S, S, t, t\rangle \rightsquigarrow^* \langle l_1, C, x, x+y\rangle$ immediately finds the solution $x = 0, l_1 = S, y = 1+w$ for some $w : Nat$, which contradicts Lemma (5).

Fixing the error in the lemma amounts to adding the hypothesis $t_1 > 0$. The fixed lemma is indeed provable, but now, the new lemma does not solve by itself the subgoal (4), for which it was posed in the first place! To deal with this problem, the whole proof has to be re-thought, and a possible solution is to prove that states of the form $\langle T, C, t_1, t_1+x+1\rangle$ such that $t_1 > 0$ are not reachable. The proof eventually succeeds, but with more effort than if the error in the lemma had been found using Maude's narrowing-based `search` command.

## 7 Conclusion, Related Work, and Future Work

State-space exploration and model checking, both enumerative and symbolic, abstraction for reducing infinite-state systems to finite ones, and interactive theorem proving for infinite-state systems are well-known verification techniques.

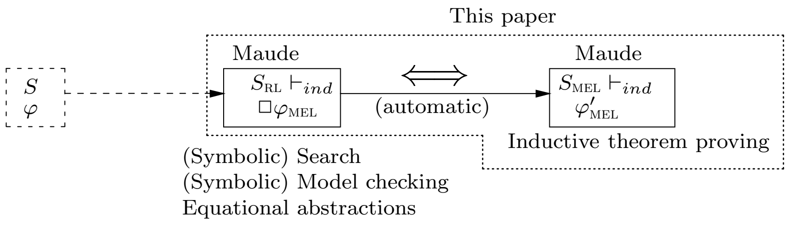

This paper
Maude
$S_{\mathrm{RL}} \vdash_{ind}$
$\Box\varphi_{\mathrm{MEL}}$
Maude
$S_{\mathrm{MEL}} \vdash_{ind}$
$\varphi'_{\mathrm{MEL}}$
$S$
$\varphi$
(automatic)
Inductive theorem proving
(Symbolic) Search
(Symbolic) Model checking
Equational abstractions

**Fig. 4.** Our approach in the context of Maude's verification tools

For rewriting-logic specifications, all but the last one are currently supported in the Maude environment. Our contribution adds a part currently missing. The approach is based on an automatic translation of invariance properties of a significant fragment of rewriting logic (topmost, without rules in conditions or frozen arguments) into inductive properties of membership equational logic. The proposed approach can then be used in conjunction with those other tools (Figure 4) thanks to the "semantical consistency" between the definition of invariance in reachable models of RL theories and the definition of narrowing. We illustrate on a simple Bakery algorithm the combination of theorem proving with narrowing for "testing" lemmas before proving them. The results are encouraging. We expect the benefits to be even more substantial for more complex systems and proofs. This statement has to be assessed by experiments. We have identified a class of systems that can be effectively symbolically simulated by narrowing, and can encode communication protocols; these are our natural future case studies.

*Related Work.* Unification and narrowing are features introduced in the latest version of Maude [11]. A tool built around Maude - the Maude NRL Analyser [17] has been used for verifying security protocols, a topic also present in [3]. We are users of (the Maude implementation of) narrowing in combination with our theorem-proving approach, and plan to use them on communication protocols.

Regarding theorem proving, our work is inspired by Bruni and Meseguer, who proposed in [16] a different encoding of RL into MEL. Their translation handles RL in its full generality, and their goal is to define the semantics and proof theory of RL in terms of those of MEL. By contrast, our encoding only captures a subset of RL, which has been shown in [3] to be expressive enough for specifying many classes of systems. But, in addition to [16] we also encode invariance properties for the given subset of RL as inductive properties in MEL, which provides us with an effective way of verifying invariants by theorem proving, possibly in interaction with Maude's other symbolic analysis tools. Moreover, our encoding is much simpler than that proposed in [16][2]. A simple encoding is essential in theorem-proving, for users to "recognise" the properties they are trying to prove.

[2] We encode reachability using only one additional *sort*, without any new operations. By contrast, [16] requires to double the number of *kinds* in the RL specification, and for each kind, there are 4 new sorts, and 4 operations defined using 7 equations each.

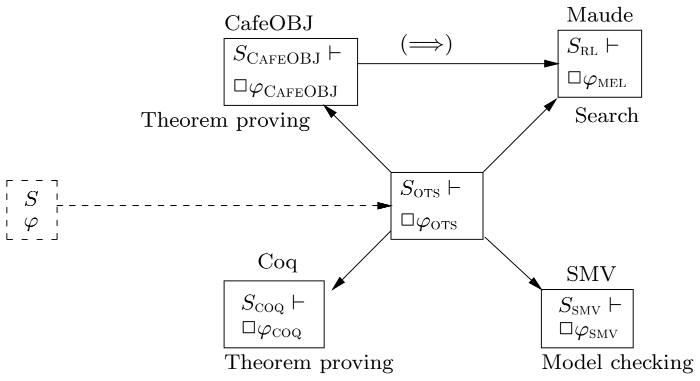

CafeOBJ
$S_{\text{CafeOBJ}} \vdash \Box\varphi_{\text{CafeOBJ}}$
$(\Longrightarrow)$
Maude
$S_{\text{RL}} \vdash \Box\varphi_{\text{MEL}}$
Theorem proving
Search
$S$
$\varphi$
$S_{\text{OTS}} \vdash \Box\varphi_{\text{OTS}}$
Coq
$S_{\text{COQ}} \vdash \Box\varphi_{\text{COQ}}$
Theorem proving
SMV
$S_{\text{SMV}} \vdash \Box\varphi_{\text{SMV}}$
Model checking

**Fig. 5.** Observational Transition Systems: representation in CafeOBJ and other tools

The present paper improves our own earlier, French version of this work [18]. The notions of dynamics and invariance that we use in the present paper are standard for systems specified in RL. Hence, our theorem-proving approach for invariants, and narrowing-based symbolic simulation for invariant falsification "talk about" the same *notion* of invariant, i.e., we have a semantical consistency. By contrast, the notions of dynamics and invariance in [18] are ad-hoc: we defined there a "ground top-level rewriting" and defined the dynamics of systems specified in RL and the notion of invariance based on that notion. Hence, in [18] we used enumerative state-space exploration for invariant falsification, without certainty that the falsification procedure deals with the *same* notion of invariant as the theorem-proving approach. Moreover, enumerative exploration can only deal with systems with finitely many initial states (expressed using finitely many ground terms). By contrast, symbolic analysis can deal with systems with possibly infinitely many initial states (expressed using a non-ground term). This is also the case of our theorem-proving approach. On the other hand, we verify in [18] a more involved, $n$-processes version of the Bakery Algorithm, where nontrivial auxiliary invariants are required for proving the mutual-exclusion goal.

There is a huge body of work dedicated to proving and disproving invariants, and it is impossible to cite all references. We limit ourselves to the approach probably closest to ours, proposed by the CafeOBJ group from Japan's Advanced Institute of Science and Technology. Their approach consists in encoding the system under verification as an Observational Transition System (OTS), and invariants as state predicates over the states of OTSs. The OTS can be represented into several formalisms (Figure 5): CAFEOBJ and Coq, for theorem proving [19,20]; Maude, for invariant falsification using enumerative techniques [21]; and SMV, for model checking [22]. Closest to our work is the theorem-proving approach in CAFEOBJ. The fundamental difference between our approach and theirs lies in the fact that we remain within one single, integrated environment and formalism (that of Maude and of rewriting logic/membership equational logic), which allows us to rely on a common semantics for the various verification activities (Figure 4). By contrast, the CafeOBJ group use several tools, with different formalisms and different underlying semantics (Figure 5). This naturally raises the question of semantical consistency. On the other hand, by not "bothering" with semantical consistency, the CafeOBJ approach can be more efficient than ours, because they can use highly-specialised tools, which are typically more efficient than Maude's (symbolic) model checker and theorem prover that we are using.

In the future we are planning to explore the integration of our theorem proving approach with Maude's symbolic model checker for temporal logic [12]. The model checker builds and analyses a symbolic graph encoding the reachable states of the system. An invariant established by theorem proving may help the symbolic model checker, by showing that certain nodes of a symbolic graph are unreachable and can be safely removed from it; thereby enabling the model checker to prove certain temporal-logic properties that could not be proved before. For example, consider a version of the Bakery algorithm containing the additional rule $\langle C, C, t_1, t_2\rangle \Rightarrow \langle C, C, t_1, t_2\rangle$, which says that if the protocol enters the critical section, it stays there forever in the same state. Assume that the symbolic graph "has" a symbolic state of the form $\langle C, C, t_1+1, t_2+1\rangle$. Then, on this graph, the temporal-logic property $\Box((t_1 > 0 \wedge t_2 > 0) \Rightarrow \Diamond(t_1 = 0 \vee t_2 = 0))$, which says that, from all reachable states state where both tickets are nonzero, a state where at least one ticket is 0 will eventually be reached, is not provable. The reason is the self-loop on $\langle C, C, t_1+1, t_2+1\rangle$. By proving mutual exclusion we can safely remove that state (and the loop responsible for the model checker's failure) from the graph, thereby possibly enabling the model checker to succeed.

## References

1. Martí-Oliet, N., Meseguer., J.: Rewriting logic: roadmap and bibliography. TCS 285(2), 121–154 (2002)
2. Meseguer, J., Rosu, G.: The rewriting logic semantics project. TCS 373(3), 213–237 (2007)
3. Meseguer, J., Thati., P.: Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. Higher-Order and Symbolic Computation 20(1-2), 123–160 (2007)
4. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sönmez, M.K.: Pathway logic: Symbolic analysis of biological signaling. In: Pacific Symposium on Biocomputing, pp. 400–412 (2002)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. L. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Ringeissen, C.: An overview of ELAN. Electr. Notes Theor. Comput. Sci. 15 (1998)
7. Diaconescu, R., Futatsugi, K.: Logical foundations of CafeOBJ. TCS 285(2), 289–318 (2002)
8. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
9. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. Electr. Notes Theor. Comput. Sci. 71 (2002)

10. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer, Heidelberg (2008)
11. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. L.: Unification and narrowing in Maude 2.4. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 380–390. Springer, Heidelberg (2009)
12. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 153–168. Springer, Heidelberg (2007)
13. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
14. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. J. Universal Computer Science 12(11), 1618–1650 (2006)
15. Escobar, S., Meseguer, J., Sasse, R.: Variant narrowing and equational unification. Electr. Notes Theor. Comput. Sci. 238(3), 103–119 (2009)
16. Bruni, R., Meseguer., J.: Semantic foundations for generalized rewrite theories. TCS 360(1-3), 386–414 (2006)
17. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Theor. Comput. Sci. 367(1-2), 162–202 (2006)
18. Rusu, V., Clavel, M.: Vérification d'invariants pour des systèmes spécifiés en logique de réécriture. In: JFLA. Studia Informatica Universalis, vol. 7.2, pp. 317–350 (2009), http://www.irisa.fr/vertecs/Equipe/Rusu/rc09.pdf
19. Futatsugi, K.: Verifying specifications with proof scores in CafeOBJ. In: ASE, pp. 3–10. IEEE Comp. Soc., Los Alamitos (2006)
20. Ogata, K., Futatsugi, K.: State machines as inductive types. IEICE Transactions 90-A(12), 2985–2988 (2007)
21. Kong, W., Seino, T., Futatsugi, K., Ogata, K.: A lightweight integration of theorem proving and model checking for system verification. In: APSEC, pp. 59–66. IEEE Comp. Soc., Los Alamitos (2005)
22. Ogata, K., Nakano, M., Nakamura, M., Futatsugi, K.: Chocolat/SMV: a translator from CafeOBJ to SMV. In: PDCAT, pp. 416–420. IEEE Comp. Soc., Los Alamitos (2005)

# Syntactic Abstraction of B Models to Generate Tests

Jacques Julliand[1], Nicolas Stouls[2],
Pierre-christophe Bué[1], and Pierre-Alain Masson[1]

[1] LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex
{bue,julliand,masson}@lifc.univ-fcomte.fr
[2] Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, France
nicolas.stouls@insa-lyon.fr

**Abstract.** In a model-based testing approach as well as for the verification of properties, B models provide an interesting solution. However, for industrial applications, the size of their state space often makes them hard to handle. To reduce the amount of states, an abstraction function can be used, often combining state variable elimination and domain abstractions of the remaining variables. This paper complements previous results, based on domain abstraction for test generation, by adding a preliminary syntactic abstraction phase, based on variable elimination. We define a syntactic transformation that suppresses some variables from a B event model, in addition to a method that chooses relevant variables according to a test purpose. We propose two methods to compute an abstraction A of an initial model M. The first one computes A as a simulation of M, and the second one computes A as a bisimulation of M. The abstraction process produces a finite state system. We apply this abstraction computation to a Model Based Testing process.

**Keywords:** Abstraction, Test Generation, (Bi-)Simulation, Slicing.

## 1 Introduction

B models are well suited for producing tests of an implementation by means of a *model-based testing* approach [1,2] and to verify dynamic properties by model-checking [3]. But model-checking as well as test generation require the models to be finite, and of tractable size. This is not usually the case with industrial applications, for which the exploration of the executions modelled frequently comes up against combinatorial explosion problems. Abstraction techniques allow for projecting the (possibly infinite or very large) state space of a system onto a small finite set of symbolic states. Abstract models make test generation or model-checking possible in practice [4]. In [5], we have proposed and experimented with an approach of test generation from abstract models. It appeared that the computation time of the abstraction could be very expensive, as evidenced by the Demoney [6] case study. We had replaced a problem of time for searching in a state graph with a problem of time for solving proofs, as the abstraction was computed by proving enabledness and reachability conditions on symbolic states [7].

In this paper, we contribute to solving this proving time problem by defining a syntactic abstraction function that requires no proof. Inspired from slicing techniques [8], the function works by suppressing some state variables from a model. In order to produce a state system that is both finite and sufficiently small, we still have to perform a semantic abstraction. This requires that some proof obligations are solved, but there are less of them than with the initial model, since it has been syntactically simplified. This approach results in semantic pruning of generated proof obligations as proposed in [9].

In Sec. 2, we introduce the notion of B event system and some of the main properties of substitution computation. Section 3 presents an Electrical System case study that illustrates our approach. In Sec. 4, we first define the set of variables to be preserved by the abstraction function and then we define the abstraction function itself. We prove that this function is correct in the sense that the generated abstract model $\mathsf{A}$ simulates or bisimulates the initial model $\mathsf{M}$. In this way, the abstraction can be used to verify safety properties and to generate tests. In Sec. 5, we present an end to end process to compute test cases from a set of observed variables by using both the semantic and the syntactic abstractions. In Sec. 6, we compare this process to a completely semantic one on several examples, and we evaluate the practical interest for test cases generation. Section 7 concludes the paper, gives some future research directions and compares our approach to other abstraction methods.

## 2 B Event Systems and Refinement

We use the B notation [10] to describe our models: this section gives the background required for reading the paper. Let us first define the following B notions: primitive forms of substitution, substitution properties and refinement. Then we will summarize the principles of before-after predicates, and conjunctive form (CF) of B predicates.

First introduced by J.-R. Abrial [11], a B event system defines a closed specification of a system by a set of events. In the sequel, we use the following notations: $x$, $x_i$, $y$, $z$ are variables and $X$, $Y$, $Z$ are sets of variables. $\mathcal{P}red$ is the set of B predicates. $I$ ($\in \mathcal{P}red$) is an invariant, and $P$, $P_1$ and $P_2$ ($\in \mathcal{P}red$) denote other predicates. The modifications of the variables are called *substitutions* in B, following [12] where the semantics of an assignment is defined as a substitution. In B, substitutions are *generalized*: they are the semantics of every kind of action, as expressed by formulas 1 to 4 below. We use $S$, $S_1$ and $S_2$ to denote B generalized substitutions, and $E$, $E_i$ and $F$ to denote B expressions. The B events are defined as generalized substitutions. All the substitutions allowed in B event systems can be rewritten by means of the five B primitive forms of substitutions of Def. 1. Notice that the multiple assignment can be generalized to $n$ variables. It is commutative, i.e. $x,\ y\ :=\ E,\ F \mathrel{\widehat{=}} y,\ x\ :=\ F,\ E$.

**Definition 1 (Substitution).** *The following five substitutions are primitive:*

- *single and multiple assignments, denoted as* $x := E$ *and* $x, y := E, F$
- *substitution with no effect, denoted as skip*
- *guarded substitution, denoted as* $P \Rightarrow S$
- *bounded nondeterministic choice, denoted as* $S_1 [] S_2$
- *substitution with local variable* $z$*, denoted as* $@z.S$.

Notice that the substitution with local variable is mainly used to express the unbounded nondeterministic choice denoted by $@z.(P \Rightarrow S)$. Let us specify that among the usual structures of specification languages, the conditional substitution IF $P$ THEN $S_1$ ELSE $S_2$ END is denoted by $(P \Rightarrow S_1)[](\neg P \Rightarrow S_2)$ with the primitive forms.

Given a substitution $S$ and a post-condition $P$, it is possible to compute the weakest precondition such that if it is satisfied, then $P$ is satisfied after the execution of $S$. The weakest precondition is denoted by $[S]P$. $[x := E]P$ is the usual substitution of all the free occurrences of $x$ in $P$ by $E$. For the four other primitive forms, the weakest precondition is computed as indicated by formulas 1 to 4 below, proved in [10].

$$[skip]P \Leftrightarrow P \tag{1}$$

$$[P_1 \Rightarrow S]P_2 \Leftrightarrow (P_1 \Rightarrow [S]P_2) \tag{2}$$

$$[S_1[]S_2]P \Leftrightarrow [S_1]P \wedge [S_2]P \tag{3}$$

$$[@z.S]P \Leftrightarrow \forall z.[S]P \qquad \text{if } z \text{ is not free in } P \tag{4}$$

$$\text{Distributivity:} \quad [S](P_1 \wedge P_2) \Leftrightarrow [S]P_1 \wedge [S]P_2 \tag{5}$$

Definition 2 defines correct B event systems. To explicitly refer to a given model, we add the name of that model as a subscript to the symbols $X$, $I$, $Init$ and $Ev$. $I_{\mathsf{M}}$ is for example the invariant of a model $\mathsf{M}$.

**Definition 2 (Correct B Event System).** *A correct B event system is a tuple* $\langle X, I, Init, Ev \rangle$ *where:*

- $X$ *is a set of state variables,*
- $I$ *(*$\in \mathcal{P}red$*) is an invariant predicate over* $X$*,*
- *Init is a substitution called* initialization*, such that the invariant holds in any initial state:* $[Init]I$*,*
- *Ev is a set of event definitions in the shape of* $ev_i \mathrel{\widehat{=}} S_i$ *such that every event preserve the invariant:* $I \Rightarrow [S_i]I$.

In Sec. 4, we will prove that an abstraction $\mathsf{A}$ that we compute is refined by its source event system $\mathsf{M}$, and so we give in Def. 3 the definition of a B event system refinement.

**Definition 3 (B Event System Refinement).** *Let* $\mathsf{A}$ *and* $\mathsf{R}$ *be two correct B event systems. Let* $I_{\mathsf{R}}$ *be their gluing invariant, i.e. a predicate that indicates how the values of the variables in* $\mathsf{R}$ *and* $\mathsf{A}$ *relate to each other.* $\mathsf{R}$ *refines* $\mathsf{A}$ *if:*

– *any initialization of* $R$ *is associated to an initialization of* $A$ *according to* $I_R$*:* $[Init_R]\neg[Init_A]\neg I_R$
– *any event* $ev \mathrel{\widehat{=}} S_R$ *of* $R$ *is an event of* $A$ *defined by* $ev \mathrel{\widehat{=}} S_A$ *in* $Ev_A$ *that satisfy* $I_R$*:* $I_A \wedge I_R \Rightarrow [S_R]\neg[S_A]\neg I_R$.

This paper also relies on two more definitions: the before-after predicate and the CF form. We denote by $Prd_X(S)$ the before-after predicate of a substitution $S$. It defines the relation between the values of the variables of the set $X$ before and after the substitution $S$. A primed variable denotes its after value. From [10], the before-after predicate is defined by:

$$Prd_X(S) \mathrel{\widehat{=}} \neg[S]\neg(\bigwedge_{x\in X}(x = x')). \tag{6}$$

**Definition 4 (Conjunctive Form).** *A B predicate* $P \in \mathcal{P}red$ *is in CF when it is a conjunction* $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ *where every* $p_i$ *is a disjunction* $p_i^1 \vee p_i^2 \vee \ldots \vee p_i^m$ *such that any* $p_i^j$ *is an elementary predicate in one of the following two forms:*

– $E(Y)\ r\ F(Z)$*, where* $E(Y)$ *and* $F(Z)$ *are B expressions on the sets of variables* $Y$ *and* $Z$ *and* $r$ *is a relational operator,*
– $\forall z.P$ *or* $\exists z.P$*, where* $P$ *is a B predicate in CF.*

Section 4 will define predicate transformation rules. We put the predicates in CF according to Def. 4 before their transformation. This allows the transformation to be correct although the negation is not monotonic w.r.t a transformation $T$ of the predicates: $T(\neg P) \neq \neg T(P)$.

## 3 Electrical System Example

We describe in this section a B event system that we will use in this paper as a running example to illustrate our proposal.

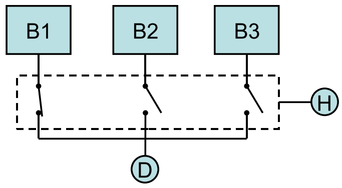


**Fig. 1.** Electrical System

A device D is powered by one of three batteries $B_1, B_2, B_3$ as shown in Fig. 1. A switch connects (or not) a battery $B_i$ to the device D. A clock H periodically sends a signal that causes a commutation of the switches, i.e. a change of the battery in charge of powering the device D. The working of the system must satisfy the three following requirements:

- $Req_1$: no short-circuit, i.e. there is only one switch closed at a time,
- $Req_2$: continuous power supply, i.e. there is always one switch closed,
- $Req_3$: a signal from the clock always changes the switch that is closed.

The batteries are subject to electrical failures. If it occurs to the battery that is powering D, the system triggers an exceptional commutation to satisfy the requirement $Req_2$. The broken batteries are replaced by a maintenance service. We assume that it works fast enough for not having more than two batteries down at the same time. When two batteries are down, the requirement $Req_3$ is relaxed and the clock signal leaves unchanged the switch that is closed.

This system is modeled in Fig. 2 by means of three variables. $H$ models the clock and takes two values: $tic$ when it asks for a commutation and $tac$ when this commutation has occurred. $Sw$ models the state of the three switches by an integer between 1 and 3: $Sw = i$ indicates that the switch $i$ is closed while the others are opened. This modelling makes that requirements $Req_1$ and $Req_2$ necessarily hold. $Bat$ models the electrical failures by a total function. The $ko$ value for a battery indicates that it is down. In addition to the typing of the variables, the invariant $I$ expresses the assumption that at least one battery is not down by stating that $Bat(Sw) = ok$. Notice that the requirement $Req_3$ is a dynamic property, not formalized in $I$. The initial state is defined by $Init$ in Fig. 2. The behavior of the system is described by four events:

- **Tic** sends a commutation command,
- **Com**[1] performs a commutation (i.e. changes the closed switch),
- **Fail** simulates an electrical failure on one of the batteries,
- **Rep** simulates a maintenance intervention replacing a down battery.

$$
\begin{array}{lll}
X & \widehat{=} & \{H,\ Sw,\ Bat\} \\
I & \widehat{=} & H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge (Bat \in 1..3 \rightarrow \{ok, ko\}) \wedge Bat(Sw) = ok \\
Init & \widehat{=} & H,\ Sw,\ Bat\ :=\ tac, 1,\ \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Tic & \widehat{=} & H = tac \Rightarrow H := tic \\
Com & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \wedge H = tic \Rightarrow \\
 & & \quad @ns.(ns \in 1..3 \wedge Bat(ns) = ok \wedge ns \neq Sw \Rightarrow H, Sw := tac, ns) \\
Fail & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
 & & \quad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow \\
 & & \qquad\quad (nb = Sw \Rightarrow @ns.(ns \in 1..3 \wedge ns \neq Sw \wedge Bat(ns) = ok \Rightarrow Sw, Bat(nb) := ns, ko)) \\
 & & \qquad [\,](nb \neq Sw \Rightarrow Bat(nb) := ko)) \\
Rep & \widehat{=} & @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat(nb) := ok)
\end{array}
$$

**Fig. 2.** B Specification of the Electrical System

## 4 Syntactic Abstraction

We define in this paper a syntactical abstraction method that applies to B models. Similar rules could be adapted for more generic formalisms such as pre-post models or transition systems.

[1] An expression $r \rhd E$ denotes a relation where the range is restricted by the set $E$. For example, $\{1 \mapsto ok,\ 2 \mapsto ko,\ 3 \mapsto ok\} \rhd \{ok\} = \{1 \mapsto ok,\ 3 \mapsto ok\}$.

Our intention is to obtain an abstract model $\mathsf{A}$ of a model $\mathsf{M}$ by observing only a subset $X_{\mathsf{A}}$ of the state variables $X_{\mathsf{M}}$ of $\mathsf{M}$. For instance, to test the electrical system in the particular cases where two batteries are down, we observe only the variable *Bat*. But to preserve the behaviors of $\mathsf{M}$ related to the variables of $X_{\mathsf{A}}$, we also keep in $\mathsf{A}$ the variables used to assign the observed variables or to define the conditions under which they are assigned.

We first present two methods to compute a set of abstract variables according to a set of observed variables. Using these variables we define a predicate and substitution transformation function. Then we describe how to compute an abstraction of a B event model $\mathsf{M}$. The abstraction is a bisimulation of $\mathsf{M}$ when the abstract variables were computed according to the second method. We also prove that if they were computed according to the first method, the abstraction is a simulation of $\mathsf{M}$.

## 4.1 Choosing the Abstract Variables

As proposed in [13], we distinguish between the *observed* variables and the *abstract* ones. A set $X_{\mathsf{A}}$ of *abstract variables* is the union of a set of *observed variables* with a set of *relevant variables*. The *Observed variables* are the ones used by the tester in a test purpose, while the *relevant variables* are the ones used to describe the evolutions of the observed variables. More precisely, the relevant variables are the ones used to assign an observed variable (*data-flow dependency*), augmented with the variables used to express when such an assignment occurs (*control-flow dependency*).

A naive method to define $X_{\mathsf{A}}$ is to syntactically collect the variables that are either on the right side or in the guard of the assignment of an observed variable. But this method will in most cases select a very large amount of variables, mainly because of the guard. For instance, if $x$ is the observed variable, then $y$ is not relevant in $(y \Rightarrow x, z := E, F)[](\neg y \Rightarrow x := E)$. A similar weakness goes for the unbounded non-deterministic choice $@z.(P \Rightarrow S)$.

Hence our contribution consists of two methods for identifying the relevant variables. The first one only considers the data-flow dependency. It is efficient, but may select a set too small of relevant variables, resulting in a set with too many behaviors in the abstracted model. The second one uses both data and control flow dependencies, but requires a predicate simplification to restrict the size of $X_{\mathsf{A}}$. It produces abstract models that have the same set of behaviors as the original model, w.r.t. the abstract variables. This second method may select a set with too many relevant variables because predicate simplification is an undecidable problem.

**Proposition 1: Data-Flow Dependency Only.** This first method considers as relevant only the variables that appear on the right side of an assignment symbol to an abstract variable. Starting from the set of observed variables, the set of all abstract variables is computed as the least fix-point when adding the relevant variables. For instance, the set of relevant variables of the electrical system is empty if the set of observed variables is $\{Bat\}$. Hence if a test purpose is only based on $Bat$, then $X_{\mathsf{A}} = \{Bat\}$. A drawback of this method is that it can introduce in $\mathsf{A}$ new execution traces w.r.t. $\mathsf{M}$. Indeed, it may weaken the guards of some of the events, that would thus become enabled more often.

**Proposition 2: Data-Flow and Control-Flow Dependencies.** This second method first computes a predicate characterizing a condition under which an abstract variable is modified, then simplifies it, and finally considers all its free variables as relevant. We express by means of formula 7 the modifications really performed by a substitution $S$ on a set $X_{\mathsf{A}}$:

$$Mod_{X_{\mathsf{A}}}(S) \mathrel{\widehat{=}} Prd_{X_{\mathsf{A}}}(S) \wedge (\bigvee_{x \in X_{\mathsf{A}}} x \neq x'). \tag{7}$$

Our intention is that the predicate, that defines the condition under which an abstract variable is modified, only involves the variables really required to modify it. Hence primed variables are not quantified, but are allowed to be free. For instance, consider $X_{\mathsf{A}} = \{x\}$ and the substitution $x := y [] (z > 0 \Rightarrow x := w) [] v := 3$. The predicate has to be in the shape of: $x' = y \vee (z > 0 \wedge x' = w)$, where the variables $y$, $w$ and $z$ are relevant whereas $v$ is not.

Finally, $X_{\mathsf{A}}$ is computed as a least fix-point, by iteratively incrementing for each event the initial set of observed variables with the relevant variables. This process terminates since the set of variables is finite. For instance, $Mod_{\{Bat\}}$ gives an empty set of relevant variables when applied to the example, as shown in Fig. 3, while $Mod_{\{H\}}$ gives $X_{\mathsf{A}} = \{Bat, H\}$.

$$\begin{array}{lll}
Mod_{\{Bat\}}(Init) & \Leftrightarrow & Bat = \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Mod_{\{Bat\}}(Tic) & \Leftrightarrow & false \text{ (no assignment of } Bat) \\
Mod_{\{Bat\}}(Com) & \Leftrightarrow & false \text{ (no assignment of } Bat) \\
Mod_{\{Bat\}}(Fail) & \Leftrightarrow & \mathrm{card}(Bat \rhd \{ok\}) > 1 \\
 & & \wedge \exists nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \wedge Bat'(nb) = ko) \\
Mod_{\{Bat\}}(Rep) & \Leftrightarrow & \exists nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \wedge Bat'(nb) = ok)
\end{array}$$

**Fig. 3.** $Mod_{\{Bat\}}$ Computation Applied to the Example

## 4.2 Predicate Transformation

Once the set of abstract variables $X_{\mathsf{A}} (\subseteq X_{\mathsf{M}})$ is defined, we have to describe how to abstract a model according to $X_{\mathsf{A}}$. We first define the transformation function $T_{X_{\mathsf{A}}}(P)$ that abstracts a predicate $P$ according to $X_{\mathsf{A}}$. We define $T_X$ on predicates in the conjunctive form (see Def. 4) by induction with the rules given in Fig. 4.

An elementary predicate is left unchanged when all the variables used in the predicate are considered in the abstraction (see the rule $R_1$). Otherwise, when an expression depends on some variables not kept in the abstraction, an elementary predicate is undetermined (see the rule $R_2$). As we want to weaken the predicate, we replace an undetermined elementary predicate by *true*. Consequently, a predicate $P_1 \wedge P_2$ is transformed into $P_1$ when $P_2$ is undetermined, and a predicate $P_1 \vee P_2$ is transformed into *true* when $P_1$ or $P_2$ is undetermined (see the rules $R_3$ and $R_4$). Finally, the transformation of a quantified predicate

$$\begin{array}{rcll}
T_X(E(Y)\ r\ E(Z)) & \widehat{=} & E(Y)\ r\ E(Z) & \text{if } Y \subseteq X \text{ and } Z \subseteq X \quad (R_1)\\
T_X(E(Y)\ r\ E(Z)) & \widehat{=} & true & \text{if } Y \not\subseteq X \text{ or } Z \not\subseteq X \quad (R_2)\\
T_X(P_1 \vee P_2) & \widehat{=} & T_X(P_1) \vee T_X(P_2) & (R_3)\\
T_X(P_1 \wedge P_2) & \widehat{=} & T_X(P_1) \wedge T_X(P_2) & (R_4)\\
T_X(\alpha z.P) & \widehat{=} & \alpha z.T_{X\cup\{z\}}(P) & (R_5)
\end{array}$$

**Fig. 4.** CF Predicate Transformation Rules

$$\begin{array}{ll}
T_{\{Bat\}}(H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge Bat \in 1..3 \rightarrow \{ok, ko\} \wedge Bat(Sw) = ok) & \\
= \begin{array}{l} T_{\{Bat\}}(H \in \{tic, tac\}) \wedge T_{\{Bat\}}(Sw \in 1..3) \\ \wedge\ T_{\{Bat\}}(Bat \in 1..3 \rightarrow \{ok, ko\}) \wedge T_{\{Bat\}}(Bat(Sw) = ok) \end{array} & \text{applying } R_4\\
= Bat \in 1..3 \rightarrow \{ok, ko\} & \text{applying } R_1 \text{ and } R_2
\end{array}$$

**Fig. 5.** Example of Predicate Transformation

is the transformation of its body w.r.t. the observed variables, augmented with the quantified variable (see the rule $R_5$).

For example the invariant $I$ of the electrical system is transformed, according to the single variable $Bat$, into $T_{\{Bat\}}(I) = Bat \in 1..3 \rightarrow \{ok, ko\}$ as in Fig. 5.

*Property 1.* Let $P$ be a CF predicate in $\mathcal{P}red$ and let $X$ be a set of variables. $P \Rightarrow T_X(P)$ is valid.

*Proof.* As we said before, $T_X(P)$ is weaker than $P$. Indeed, for any predicate $P$ in CF there exist $p_1$ and $p_2$ such that $P = p_1 \wedge p_2$ and such that it is transformed either into $p_1 \wedge p_2$, or into $p_1$, or into $p_2$, or into $true$, by application of the transformation rules $R_i$. For any disjunctive predicate $P$ there exist $p_1$ and $p_2$ such that $P = p_1 \vee p_2$ and $p_1 \vee p_2$ is transformed either into $p_1 \vee p_2$ or into $true$.

## 4.3 Substitution Transformation

The abstraction of substitutions is defined through cases in Fig. 6 on the primitive forms of substitutions. Intuitively, any assignment $x := E$ is preserved into the transformed model if and only if $x$ is an abstract variable. According to both of the two methods described in sec. 4.1, if $x$ is an abstract variable, then so are all the variables in $E$. Therefore, in rules $R_6$ to $R_{11}$, we do not transform the expressions $E$ and $F$.

A substitution is abstracted by *skip* when it does not modify any variable from $X$ (see rules $R_6$, $R_8$, $R_9$ and $R_{10}$ in which $y := F$ is abstracted by *skip*). The assignment of a variable $x$ is left unchanged if $x$ is an abstract variable (see rules $R_7$, $R_{10}$, $R_{11}$). The transformation of a guarded substitution $S$ is such that $T_X(S)$ is enabled at least as often as $S$, since $T_X(P)$ is weaker than $P$ from Prop. 1 (see rule $R_{12}$). The bounded non deterministic choice $S_1 [] S_2$ becomes a bounded non deterministic choice between the abstraction of $S_1$ and $S_2$ (see rule $R_{13}$). The quantified substitution is transformed by inserting the bound variable into the set of abstract variables (see rule $R_{14}$).

$$
\begin{array}{rcllr}
T_X(x \ := \ E) & \widehat{=} & skip & \text{if } x \notin X & (R_6)\\
T_X(x \ := \ E) & \widehat{=} & x \ := \ E & \text{if } x \in X & (R_7)\\
T_X(skip) & \widehat{=} & skip & & (R_8)\\
T_X(x,\ y := \ E,\ F) & \widehat{=} & skip & \text{if } x \notin X \text{ and } y \notin X & (R_9)\\
T_X(x,\ y := \ E,\ F) & \widehat{=} & x \ := \ E & \text{if } x \in X \text{ and } y \notin X & (R_{10})\\
T_X(x,\ y := \ E,\ F) & \widehat{=} & x,\ y := \ E,\ F & \text{if } x \in X \text{ and } y \in X & (R_{11})\\
T_X(P \Rightarrow S) & \widehat{=} & T_X(P) \Rightarrow T_X(S) & & (R_{12})\\
T_X(S_1[]S_2) & \widehat{=} & T_X(S_1)[]T_X(S_2) & & (R_{13})\\
T_X(@z.S) & \widehat{=} & @z.T_{X\cup\{z\}}(S) & & (R_{14})
\end{array}
$$

**Fig. 6.** Primitive Substitution Transformation Rules

## 4.4 B Event System Transformation

According to the predicate and substitution transformation functions (see figure 4 and figure 6), we define the transformation of a B event model according to a set of abstract variables (section 4.1) in Def. 5. This transformation translates a correct model $\mathsf{M}$ into a model $\mathsf{A}$ that simulates $\mathsf{M}$ (Sec. 4.5). The electrical system is transformed as shown in Fig. 7 for the set of abstract variables $\{Bat\}$.

**Definition 5 (B Event System Transformation).** *Let $X_\mathsf{A}$ be a set of abstract variables, defined as in Sec. 4.1 from a set of observed variables $X$ with $X \subseteq X_\mathsf{M}$. A correct B event system $\mathsf{M} = \langle X_\mathsf{M}, I_\mathsf{M}, Init_\mathsf{M}, Ev_\mathsf{M}\rangle$ is abstracted as the B event system $\mathsf{A} = \langle X_\mathsf{A}, I_\mathsf{A}, Init_\mathsf{A}, Ev_\mathsf{A}\rangle$ as follows:*

- *$X_\mathsf{A} \subseteq X_\mathsf{M}$, the set of abstract variables is a subset of the state variables,*
- *$I_\mathsf{A} = T_{X_\mathsf{A}}(I_\mathsf{M})$, the invariant is transformed,*
- *$Init_\mathsf{A} = T_{X_\mathsf{A}}(Init_\mathsf{M})$, the initialization is transformed,*
- *to each event $ev \mathrel{\widehat{=}} S_\mathsf{M}$ in $Ev_\mathsf{M}$ is associated $ev \mathrel{\widehat{=}} T_{X_\mathsf{A}}(S_\mathsf{M})$ in $Ev_\mathsf{A}$.*

## 4.5 Correctness

When the set of abstract variables $X_\mathsf{A}$ preserve both the data and control flows as defined in Sec. 4.1 (Proposition 2), the transition relation, restricted to $X_\mathsf{A}$, is preserved, as proved by theorem 1. $\mathsf{A}$ and $\mathsf{M}$ have an equivalent before-after relation $Prd_{X_\mathsf{A}}$, therefore they are bisimilar. Hence when a CTL* property is verified on $\mathsf{A}$ it holds on $\mathsf{M}$ and test cases generated from $\mathsf{A}$ can always be instantiated on $\mathsf{M}$.

**Theorem 1.** *Let $S$ be a substitution. Let $X$ be a set of abstract variables composed of any free variable of $Mod_X(S)$, we have $Prd_X(S) \Leftrightarrow Prd_X(T_X(S))$.*

With the method defined in Sec. 4.1 by Proposition 1, $\mathsf{A}$ is a simulation of $\mathsf{M}$. The B refinement relation (see Def. 3) is proven in [14] to be a simulation: $\mathsf{A}$ simulates $\mathsf{M}$ by a $\tau$-simulation. $\tau$ is a silent action corresponding in our case to an event reduced to *skip* or to $P \Rightarrow skip$. Theorems 2 and 3 establish that $\mathsf{M}$ refines $\mathsf{A}$, and thus that $\mathsf{A}$ simulates $\mathsf{M}$. The safety properties are preserved, but some tests generated from $\mathsf{A}$ might be impossible to instantiate on $\mathsf{M}$.

**Theorem 2.** *Let $I$ be a CF invariant of a correct B event system, let $S$ be a substitution and let $X$ be a set of abstract variables. The transformation rules $R_6$ to $R_{14}$ are such that $S$ refines $T_X(S)$ according to the invariant $I$.*

**Theorem 3.** *Let $X$ be a set of abstract variables defined as in Proposition 1. Let $T_X$ be the transformation defined in Fig. 6, and let $\mathsf{A}$ be an abstraction of an event system $\mathsf{M}$ defined according to Def. 5. $\mathsf{A}$ is refined by $\mathsf{M}$ in the sense of Def. 3.*

Theorem 2 establishes that any substitution $S$ refines its transformation $T_X(S)$ for a given set of abstract variables $X$. Theorem 3 establishes that a B event system $\mathsf{M}$ refines the B abstract system obtained according to Def. 5 by applying to $\mathsf{M}$ the transformation rules of Fig. 4 and Fig. 6.

*Proof (of theorem 3).* This is a direct consequence of theorem 2 and Def. 5 since the substitution $Init_{\mathsf{A}} \mathrel{\widehat{=}} T_X(Init_{\mathsf{M}})$ is refined by $Init_{\mathsf{M}}$, and that for any event $ev \mathrel{\widehat{=}} S_{\mathsf{M}}$, the substitution $S_{\mathsf{A}} \mathrel{\widehat{=}} T_X(S_{\mathsf{M}})$ is refined by $S_{\mathsf{M}}$.

$$
\begin{array}{lll}
X & \widehat{=} & \{Bat\} \\
I & \widehat{=} & Bat \in 1..3 \rightarrow \{ok, ko\} \\
Init & \widehat{=} & Bat := \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Tic & \widehat{=} & skip \\
Com & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow @ns.(ns \in 1..3 \wedge Bat(ns) = ok \Rightarrow skip) \\
Fail & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
 & & \quad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow Bat(nb) := ko) \\
Rep & \widehat{=} & @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat(nb) := ok)
\end{array}
$$

**Fig. 7.** B Syntactically Abstracted Specification of the Electrical System

## 5 Application of the Method to a Testing Process

We show in this section how to use the syntactic abstraction in a model-based testing approach.

### 5.1 Test Generation from an Abstraction

We have described in [5] a model-based testing process using an abstraction as input. It can be summarized as follows. A validation engineer describes by means of a handwritten test purpose $\mathsf{TP}$ how he intends to test the system, according to his know-how. We have proposed in [15] a language based on regular expressions, to describe a $\mathsf{TP}$ as a sequence of actions to fire and states to reach (targeted by these actions). The actions can be explicitly called in the shape of event names, or left unspecified by the use of a generic name. The unspecified calls then have to be replaced with explicit event names. However, a combinatorial explosion problem occurs, when searching in a concrete model for the possible replacements that lead to the target states. This leads us to use abstractions instead of concrete models. Figure 8 shows our approach.

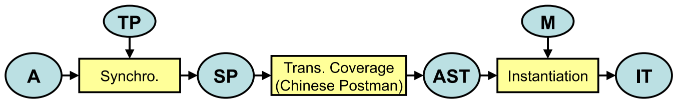
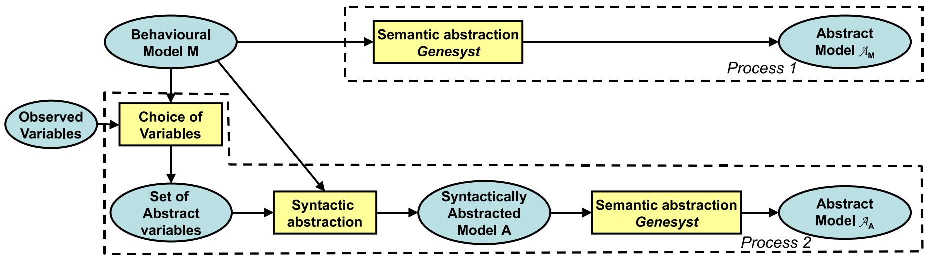


**Fig. 8.** Generating Tests from Test Purpose by Abstraction



Behavioural Model M
Semantic abstraction *Genesyst*
Abstract Model $\mathcal{A}_M$
Process 1
Observed Variables
Choice of Variables
Set of Abstract variables
Syntactic abstraction
Syntactically Abstracted Model A
Semantic abstraction *Genesyst*
Abstract Model $\mathcal{A}_A$
Process 2

**Fig. 9.** Abstraction Process

We perform a synchronized product between an abstraction A and the automaton of a TP. This results in a model SP whose executions are the executions of A that match the TP. An implementation [16] of the Chinese Postman algorithm is applied to SP to cover its transitions. The result is a set of abstract symbolic tests AST. These tests are instantiated from M as a set IT of instantiated tests.

## 5.2 Abstraction Computation

We show in this section two ways of producing an abstraction A that can be used as an input of the process of Fig. 8. The syntactic abstraction of Sec. 4 is used in one of these two ways.

In order to compute the synchronized product of an abstraction A with the automaton of a TP, we compute the semantics of A as a labelled transition system. We use *GeneSyst* [7] for that purpose. This tool computes a semantic abstraction of a B model in the shape of a symbolic labelled transition system. The semantic abstraction relies on feasibility proofs of the transitions between two symbolic states. *GeneSyst* generates proof obligations (POs) for each of the potential transitions between two symbolic states, and tries to solve them automatically.

The two main drawbacks of this process are its time cost and the proportion of POs not automatically solved. Indeed, each unsolved PO results in a transition that is kept in the symbolic labelled transition system, although it is possibly unfeasible. An abstract symbolic test going through such a transition may be impossible to instantiate from the concrete model M. By applying a preliminary phase of syntactic abstraction, we reduce the impact of that problem by reducing the number and the size of the POs, since *GeneSyst* operates on an already abstracted model. For example, no proof obligation is generated for an event reduced to *skip* (it becomes a reflexive transition on any symbolic state).

The experimental results presented in Sec. 6 compare two approaches. The first one (see Fig. 9/Process 1) is only semantic, while the second one (see Fig. 9/Process 2) combines a syntactic and a semantic abstraction.

## 6 Experimental Results

We have applied our method to four case studies. They are various cases of reactive systems: an automatic conveying system (Robot [17]), a reverse phone book service (Qui-Donc [2]), the electrical system[2] (Electr.) and an electronic purse (DeMoney [6]). Each one is abstracted w.r.t. two sets of abstract variables. These sets have been computed according to Proposition 1 of Sec. 4.1. We also have tried to compute the abstract variables according to Proposition 2, but all the variables have been computed as *abstract* in three case studies. Only for the electrical system the set of abstract variables was the same as with Proposition 1. These case studies reveal a limit in the application of Proposition 2.

In Sec. 6.1 we present an experimental evaluation of the syntactic abstraction. Then, in Sec. 6.2 we compare $\mathcal{A}_{\mathsf{M}}$ with $\mathcal{A}_{\mathsf{A}}$ respectively computed by the semantic abstraction process or by its combination with the syntactic one.

### 6.1 Impact of the Syntactic Abstraction on Models

Table 1 indicates the size of the case studies and the syntactically abstracted models. The Symbols "♯", "Ev.", "Var." and "Pot." respectively stand for *number of*, *Events*, *Variables* and *Potential*. For example the Robot, defined by 9 events and 6 variables is abstracted w.r.t. two sets of respectively 3 and 4 abstract variables.

**Table 1.** Size of the Case Studies and of their Syntactical Abstractions

| Case Study | ♯Ev. | Model M | | | Syntactically abstracted model A | | | |
|---|---|---|---|---|---|---|---|---|
| | | ♯Var. | ♯B lines | ♯Pot. states | ♯Var. | ♯B lines | ♯Pot. states | ♯Symb. states |
| Robot | 9 | 6 | 100 | 384 | 3 | 90 | 48 | 6 |
| | | | | | 4 | 90 | 144 | 8 |
| QuiDonc | 4 | 3 | 170 | 13 | 2 | 160 | 16 | 5 |
| | | | | | 2 | 160 | 16 | 6 |
| Electr. | 4 | 3 | 100 | 36 | 1 | 50 | 5 | 2 |
| | | | | | 1 | 40 | 2 | 2 |
| DeMoney | 11 | 9 | 330 | $10^{30}$ | 1 | 140 | 65536 | 3 |
| | | | | | 2 | 180 | 7 | 4 |

A direct observable result of the syntactic abstraction is a reduction of the number of potential states of the model. Also notice that the simplification reduces from 10% up to 50% the number of lines of the model.

[2] The 100 lines length of the model, in Table 1, refer to a "verbose" version of the model, much more readable than our version of Fig. 2.

## 6.2 Comparison of the Abstraction Processes 1 and 2

Table 2 compares the abstractions computed either directly from the behavioral models (see process 1 in Fig. 9), or from their syntactic abstractions (see process 2 in Fig. 9). The abbreviations "Trans.", "Unau.", "Inst." and "Cover." stand respectively for *transitions*, *unauthorized*, *instantiated* and *coverage*.

**Table 2.** Comparison of the semantic and syntactic/semantic abstraction processes

| Case study | Process 1 : $\mathcal{A}_{\mathsf{M}}$ | | | | | | Process 2 : $\mathcal{A}_{\mathsf{A}}$ | | | | | | Traces inclusion |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ♯Trans. | ♯Unau. Trans. | ♯PO | Time (s) | ♯Inst./ ♯Tests. | Trans. Cover. of $\mathcal{A}_{\mathsf{M}}$ | ♯Trans. | ♯Unau. Trans. | ♯PO | Time (s) | ♯Inst./ ♯Tests. | Trans. Cover. of $\mathcal{A}_{\mathsf{A}}$ | |
| Robot | 42 | 5 | 263 | 64 | 4/11 | 29/37 (78%) | 36 | 0 | 143 | 35 | 7/11 | 31/36 (86%) | $\mathcal{A}_{\mathsf{A}} \subset \mathcal{A}_{\mathsf{M}}$ |
| | 51 | 0 | 402 | 76 | 4/23 | 35/51 (68%) | 50 | 0 | 242 | 49 | 8/23 | 38/50 (76%) | $\mathcal{A}_{\mathsf{A}} \subset \mathcal{A}_{\mathsf{M}}$ |
| Qui- | 20 | 2 | 71 | 19 | 9/11 | 12/18 (66%) | 25 | 7 | 89 | 21 | 6/11 | 11/18 (61%) | $\mathcal{A}_{\mathsf{A}} \not\subseteq \mathcal{A}_{\mathsf{M}}$ |
| Donc | 25 | 2 | 89 | 21 | 4/10 | 6/23 (26%) | 29 | 6 | 103 | 23 | 4/10 | 6/23 (26%) | $\mathcal{A}_{\mathsf{A}} \not\subseteq \mathcal{A}_{\mathsf{M}}$ |
| Electr. | 13 | 5 | 26 | 7 | 2/2 | 8/8 (100%) | 13 | 5 | 16 | 5 | 2/2 | 8/8 (100%) | $\mathcal{A}_{\mathsf{A}} = \mathcal{A}_{\mathsf{M}}$ |
| | 7 | 0 | 21 | 5 | 3/3 | 7/7 (100%) | 7 | 0 | 9 | 2 | 3/3 | 7/7 (100%) | $\mathcal{A}_{\mathsf{A}} = \mathcal{A}_{\mathsf{M}}$ |
| De- | 38 | 5 | 116 | 189 | 17/18 | 25/33 (76%) | 38 | 5 | 68 | 38 | 17/18 | 25/33 (76%) | $\mathcal{A}_{\mathsf{A}} \subset \mathcal{A}_{\mathsf{M}}$ |
| Money | 53 | 0 | 290 | 172 | 22/38 | 30/53 (56%) | 50 | 0 | 130 | 65 | 20/35 | 26/50 (52%) | $\mathcal{A}_{\mathsf{A}} \subset \mathcal{A}_{\mathsf{M}}$ |

We see on our examples that there is between 1.8 and 2.3 fewer POs to compute with process 2 than with process 1, except for the Qui-Donc. The semantic abstraction computation in process 2 takes from twice up to five times less time than in process 1, where no previous syntactic abstraction have been performed. For the Qui-Donc, the syntactical abstraction has too much over-approximated the initial model, which explains the augmentation of the POs w.r.t. the process 1. Finally, there are four cases out of eight where the abstraction $\mathcal{A}_{\mathsf{A}}$ is more precise than $\mathcal{A}_{\mathsf{M}}$ in the sense that it has less transitions, due to the reduction of the number of unproved POs. In these four cases, the set of traces of $\mathcal{A}_{\mathsf{A}}$ is included in the set of traces of $\mathcal{A}_{\mathsf{M}}$. In the case of the electrical system, the set of traces are equal. In the Qui-Donc case, the traces cannot be compared. The simplification by the syntactic abstraction of the events and of the invariant makes that $\mathcal{A}_{\mathsf{A}}$ may contain more transitions (thus more traces) than $\mathcal{A}_{\mathsf{M}}$. But the number and the difficulty of the POs is greater to get $\mathcal{A}_{\mathsf{M}}$ than to get $\mathcal{A}_{\mathsf{A}}$, so that proof failures may occur more often with $\mathcal{A}_{\mathsf{M}}$. As a result, $\mathcal{A}_{\mathsf{M}}$ can also contain transitions that are not in $\mathcal{A}_{\mathsf{A}}$.

As for the ratios of tests instantiated and of transitions covered of the abstraction, we observe their stability with or without syntactic abstraction. Although the ratios are a bit better (or equal) for the Robot and the Electrical System, and a bit worse for Qui-Donc and Demoney, they are mainly very close to each other. But, due to the reduction of the number of POs, the time to obtain these comparable results is improved with process 2, i.e. when there is a preliminary syntactic abstraction phase. Again, this is not true for the Qui-Donc since on the contrary, its number of POs has increased.

Finally, the method had no interest with the Qui-Donc, which was the smallest example. But, as shown by DeMoney, its efficiency in terms of gain of the abstraction computation time, of reduction of the number of unproved POs and of precision of the abstraction, grows with the size of the examples.

## 7 Conclusion, Related Works and Further Works

We have presented in the B framework a method for abstracting an event system by elimination of some state variables. In this context, we have proposed two methods to compute the set of variables kept in the abstraction according to the set of observed variables. We have proved that when using the first method, the generated abstraction simulates the concrete model, while when using the second method, the generated abstraction bi-simulates the concrete model. This is useful for verifying safety properties and generating tests.

In the context of test generation, our method consists in initializing the test generation process from event B model described in [5], by a syntactic abstraction. Since the syntactic abstraction reduces the size of the model, the main advantage of this method is that it reduces the set of uninstantiable tests, by reducing the level of abstraction (reduces the number of PO generated and facilitates the proof of the remaining PO). Moreover, this results in a gain of computation time. We believe that the bigger the ratio of the number of state variables to the number of observed variables is, the bigger the gain is. This conjecture needs to be confirmed by experiments on industrial size applications.

Many other works define model abstraction methods to verify properties or to generate tests. The method of [18] uses an extension of the model-checker Mur$\phi$ to compute tests from projected state coverage criteria that eliminate some state variables and project others on abstract domains. In [19], an abstraction is computed by partition analysis of a state-based specification, based on the pre and post conditions of the operations. Constraint solving techniques are used. The methods of [20,21,22] use theorem proving to compute the abstract model, which is defined over boolean variables that correspond to a set of *a priori* fixed predicates. In contrast, our method first introduces a syntactical abstraction computation from a set of observed variables, and further abstracts it by theorem proving. [23] also performs a syntactic transformation, but requires the use of a constraint solver during a model checking process.

Other automatic abstraction methods [24] are limited to finite state systems. The deductive model checking algorithm of [25] produces an abstraction w.r.t. a LTL property by an iterative refinement process that requires human expertise. Our method can handle infinite state space specifications. The paper [26] presents a syntactic abstraction method for guarded command programs based on assignment substitution. The method is sound and complete for programs without unbounded non determinism. However, the method is iterative and does not terminate in the general case. It requires the user to give an upper-bound of the number of iterations. The paper also presents an extension for unbounded non deterministic programs that is sound but not complete, due to an exponential number of predicates generated at each iteration step. In contrast, our syntactic method is iterative on the syntactic structure of the specifications. It is sound but not complete. It handles unbounded non deterministic specifications with no need for other iterative process and always terminates. Above all, our method does not compute any weakest precondition whereas the approach in [26] does, which possibly introduces infinitely many new predicates.

The syntactic method that we have presented is correct, but, in the case of Proposition 1, may sometimes produce inaccurate over-approximations due to a too strong abstraction (see for example the experiments on the Qui-Donc). Proposition 2 produces a bisimulation, but may leave the initial model unchanged, i.e. not abstracted, if all the variables are computed as abstract. We have to find a compromise between the two propositions, that would reduce the number of abstract variables, but that would keep at least partially the control structure of the operations. Also, we think that rules could be improved to get a finer approximation. For instance, improving the rules is possible when the invariant contains an equivalence such as $x = c \Leftrightarrow y = c'$. If $y$ is an eliminated variable and $x$ an observed one, we could substitute all the occurrences of the elementary predicate $y = c'$ with $x = c$. This would preserve the property in the syntactic abstraction $\mathcal{A}_{\mathsf{A}}$, so that the following semantic abstraction would be more accurate. Such rules should prevent the addition of transitions in the Qui-Donc abstraction $\mathcal{A}_{\mathsf{A}}$ w.r.t. $\mathcal{A}_{\mathsf{M}}$.

We think that extending the test generation method introduced in [5] by using a combination of syntactic and semantic abstractions will improve the method, since the abstraction is more accurate if there are less unproved POs.

## References

1. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
2. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science, Amsterdam (2006)
3. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. Software Tools for Technology Transfer 10(2), 185–203 (2008)
4. Bouquet, F., Couchot, J.F., Dadeau, F., Giorgetti, A.: Instantiation of parameterized data structures for model-based testing. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 96–110. Springer, Heidelberg (2006)
5. Bouquet, F., Bué, P.C., Julliand, J., Masson, P.A.: Test generation based on abstraction and test purposes to complement structural tests. In: A-MOST 2010, 6th int. Workshop on Advances in Model Based Testing, Paris, France (April 2010)
6. Marlet, R., Mesnil, C.: Demoney: A demonstrative electronic purse. Technical Report SECSAFE-TL-007, Trusted Logic (2002)
7. Bert, D., Potet, M.L., Stouls, N.: GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. In: Treharne, H., King, S., Henson, M. C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 299–318. Springer, Heidelberg (2005)
8. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering SE-10(4), 352–357 (1984)
9. Couchot, J.F., Giorgetti, A., Stouls, N.: Graph-based Reduction of Program Verification Conditions. In: AFM 2009 (2009)
10. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
11. Abrial, J.R.: Extending B without changing it (for developing distributed systems). In: 1st B Conference, pp. 169–190 (1996)

12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 10(12), 576–580 (1969)
13. Brückner, I., Wehrheim, H.: Slicing an Integrated Formal Method for Verification. In: Lau, K. K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 360–374. Springer, Heidelberg (2005)
14. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Ready-simulation is not ready to express a modular refinement relation. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 266–283. Springer, Heidelberg (2000)
15. Julliand, J., Masson, P.A., Tissot, R.: Generating security tests in addition to functional tests. In: AST 2008, pp. 41–44. ACM Press, New York (2008)
16. Thimbleby, H.: The directed chinese postman problem. Software: Practice and Experience 33(11), 1081–1096 (2003)
17. Bouquet, F., Bué, P.C., Julliand, J., Masson, P.A.: Génération de tests à partir de critères dynamiques de sélection et par abstraction. In: AFADL 2009, Toulouse, France, January 2009, pp. 161–176 (2009)
18. Friedman, G., Hartman, A., Nagin, K., Shiran, T.: Projected state machine coverage for software testing. In: ISSTA, pp. 134–143 (2002)
19. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Larsen, P.G., Woodcock, J.C.P. (eds.) FME 1993. LNCS, vol. 670, pp. 268–284. Springer, Heidelberg (1993)
20. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
21. Bensalem, S., Lakhnech, Y., Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)
22. Colon, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)
23. Chan, W., Anderson, R., Beame, P., Notkin, D.: Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-Linear Constraints. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
24. Clarke, E., Grumberg, O., Long, D.: Model Checking and Abstraction. TOPLAS 1994, ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)
25. Sipma, H., Uribe, T., Manna, Z.: Deductive model checking. Formal Methods in System Design 15(1), 49–74 (1999)
26. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000)

# Building a Test-Ready Abstraction of a Behavioral Model Using CLP

Pierre-Christophe Bué, Frédéric Dadeau,
Adrien de Kermadec, and Fabrice Bouquet

Laboratoire d'Informatique – Université de Franche-Comté
LIFC/INRIA CASSIS Project – 16 route de Gray – 25030 Besançon cedex, France
{pcbue,fdadeau,adekermadec,fbouquet}@lifc.univ-fcomte.fr

**Abstract.** This paper proposes an approach for automatically generating model-based tests from a symbolic transition system built as an abstraction of a textual model description written using a pre/postcondition formalism. The abstraction gathers into equivalence classes states from which the same set of operation behaviors can be activated, computed using constraint solving techniques. This abstraction is then used to generate model-based tests using an online test generation technique in which the model animation is guided by the exploration of the abstraction. We apply this approach on the B abstract machines formalism, and compare with a commercial tool, named Leirios Test Generator. We show that our approach makes it possible to achieve a wider variety of test cases, by exploiting the knowledge of the model topology, resulting in an improved fault detection capability.

## 1 Introduction

In the context of Model-Based Testing [2], a formal model is used for both computing the test cases and the oracle, i.e., the expected results obtained when the tests are played on the System Under Test (SUT). Nowadays, two main approaches are used for modelling the SUT. The first one is to use a pre/postcondition notation, generally textual, that describes the system using state variables and a transition relation that updates their values, as in B [1] or VDM [14]. The second solution is to use a graph-based description of the states and transitions of the SUT, providing a topology of the state space, as in the FSM [15] or IOLTS [20] formalisms. While the first approach unburdens the validation engineer from drawing the complete state/transition graph, it is more difficult to use for automated test cases computation. To overcome this problem, many approaches bridge the gap between these two kinds of formalisms by building a graph representation of the system, by simulating the execution of the model. For example, ProTest [19] builds a labeled transition system representing a B model, using the ProB animator [16]. Unfortunately, such a technique is limited in presence of large state space systems, that involve sets or ranges of integers, which may cause a dramatic increase of the number of states. To overcome this limitation, other approaches rely on the use of abstractions [6], or symbolic techniques and formalisms, such as IOSTS [7], that avoid the complete enumeration of the system states, by gathering them into symbolic states.

The objective of our work is to propose a technique for automatically building an abstraction of a system, using symoblic techniques. We apply this technique on behavioral models, namely B abstract machines, in order to produce a symbolic transition system (STS) that will make it possible to improve a test generation process by providing a helpful state/transition reachability analysis, while keeping the STS of a tractable size.

Our approach relies on the notion of behaviors which results from the decomposition of the operations regarding their possible executions flows, originally introduced by Dick and Faivre in [11]. We assume that our models are deterministic, meaning that an invocation of an operation leads to the activation of (at most) one behavior. The STS is then built in two steps. We first define each symbolic state by the set of behaviors that can (and can not) be activated from it. Then we compute, using constraint solving techniques, the transitions between symbolic states as the behaviors that can lead from a state to another. Once this STS has been computed, we apply on it "classical" exploration algorithms that aim at producing the model-based test cases. Since the STS is an over-approximation of the original B model, in which certain cases of non-determinism may remain, we propose to perform an *online* test generation by coupling the exploration of the graph with the animation of the model, so as to have a direct feedback on the exploration progress. The generated tests can then be concretized to be played on the System/Implementation Under Test.

The paper is organized as follows. Section 2 presents the behavioral models that we consider, namely the B abstract machines formalism, and introduces useful definitions. Then, Section 3 describes the process used to produce the STS representation of the B model. The test generation principles that we use are presented in Sect. 4. We describe a tool-supported experimentation of this method in Sect. 5. We discuss related works in Sect. 6. Finally, Section 7 concludes and presents the future works.

## 2 Behavioral Models

This section defines the notion of behavioral models, declined on the B abstract machines formalism. We present the concept of behaviors, computed from the operations of the B model. Finally, we introduce some definitions that will be used throughout the remainder of the paper.

### 2.1 The B Abstract Machines

The B method [1] is dedicated to formal development, from high-level specification to implementable code. Specifications are based on three formalisms: data are specified using a set theory, properties are first-order predicates and the behavioral part is specified by *Generalized Substitutions*.

The B method starts by the writing of a formal specification, named *abstract machine*, that gives a functional view of the system. The machine is then spiced up with invariant properties that represent properties that have to hold at each state of the system execution. It means that (*i*) the initialization has to establish the invariant, (*ii*) the operations have to preserve the invariant (meaning that if the invariant is satisfied before the operation, then it also has to be satisfied after the execution of the operation). Operations are written in terms of Generalized Substitutions that are built on basic assignments, composed into more generalized and expressive structures, that may for example, represent conditional substitutions (IF...THEN...ELSE...END) or non-deterministic constructs (CHOICE, ANY).

**Running example of a B Abstract Machine.** The B machine presented in Fig. 1 models a simple process scheduler, initially introduced in [11] that manages the access of processes (contained in the abstract set `PID`) to a critical section. Three sets represent the different states of the processes, that can be idle (`waiting` variable), in the waiting queue for the critical section (`ready` variable), or accessing the critical section (`active` variable). As a structural constraint, only one process at the time can access the critical section, and no process can wait for a free critical section. The model presents five operations. `new` creates a non-existing process and puts it in the idle state. `delete` removes an existing idle process from the system. `ready` makes its parameter access the critical section (if empty) or places the process in the waiting queue. The `swap` operation removes the process currently using the critical section and places

```
MACHINE
  scheduler

SETS
  PID = {p1,p2,p3,p4,p5}

VARIABLES
  active, ready, waiting

INVARIANT
  active ⊆ PID ∧ ready ⊆ PID ∧
  waiting ⊆ PID ∧ ready ∩ waiting = ∅ ∧
  active ∩ waiting = ∅ ∧
  active ∩ ready = ∅ ∧ card(active) ≤ 1 ∧
  (active = ∅ ⇒ ready = ∅)

INITIALIZATION
  active := ∅ || ready := ∅ || waiting := ∅

OPERATIONS
  new(pp) ≙
    PRE pp ∈ PID ∧
        pp ∉ (waiting ∪ ready ∪ active)
    THEN waiting := waiting ∪ {pp}
    END;

  del(pp) ≙
    PRE pp ∈ waiting
    THEN waiting := waiting - {pp}
    END;

  ready(pp) ≙
    PRE pp ∈ waiting THEN
      waiting := waiting - {pp} ||
      IF active = ∅ THEN active := {pp}
      ELSE ready := ready ∪ {pp}
      END
    END;

  swap ≙
    PRE active ≠ ∅ THEN
      waiting := waiting ∪ active ||
      IF ready = ∅ THEN active := ∅
      ELSE ANY pp WHERE pp ∈ ready THEN
              active := {pp} ||
              ready := ready - {pp}
           END
      END
    END;

  rr ← observe(pp) ≙
     PRE pp ∈ 1..3 THEN
     THEN IF pp = 1 THEN rr := waiting
          ELSE IF pp = 2 THEN rr := ready
               ELSE rr := active
               END
          END
     END
END
```

**Fig. 1.** B abstract machine of a simple process Scheduler

**Table 1.** Rules to compute the behaviors from an operation

| Substitution | Rewriting |
|---|---|
| Bounded choice | $Behav_x(S \; [] \; T) \rightsquigarrow Behav_x(S) \; [.] \; Behav_x(T)$ |
| Precondition | $Behav_x(P \mid S) \rightsquigarrow P \; \wedge \; Behav_x(S)$ |
| Guarded | $Behav_x(P \Longrightarrow S) \rightsquigarrow P \; \wedge \; Behav_x(S)$ |
| Unbounded choice | $Behav_x(@z.S) \rightsquigarrow Behav_x(S)$ and $z \in L$ |
| Others | $Behav_x(S) \rightsquigarrow prd_x(S)$ for $S \in \{skip, x := E, x :\in E\}$ |

it in the idle state; it then chooses a process from the waiting queue to give it access to the critical section. Finally, the `observe` operation is used for testability purposes, and retrieves the values of the states variables.

## 2.2 Behavioral Models

The behaviors of a $B$ operation (a generalized substitution) are described by before-after predicates that represent the possible ways of executing the operation, in terms of activated effect. The computation of the behaviors from a $B$ operation *op* is the result of the rule $Behav(S)$, given in Tab. 1, completed by the rules of [1, p.309] (for handling parallel substitutions).

In this table, $P$ is a predicate, $S$ and $T$ are generalized substitutions, [.] is the bounded choice operator between behaviors, $L$ is a vector of local variables of the operation, $V$ is the vector of machine state variables on which the predicates apply, $prd_x$ is the transformation of an assignment into a before-after predicate (e.g. $prd_x(x := E) \Leftrightarrow x' = E$). Bounded choice and guarded substitutions are generally used together to express IF $P$ THEN $S$ ELSE $T$ END substitutions, written $(P \Longrightarrow S \; [] \; \neg P \Longrightarrow T)$ in this formalism.

**Definition 1 (Elementary Behavior Normal Form).** *An operation is in Elementary Behavior Normal Form (EBNF) when it is expressed as a bounded choice between behaviors in which each elementary behavior predicate b is expressed as* $\langle StateP^b(V),\; InP^b(V, I^b),\; LocP^b(V, I^b, L^b),\; E\!f\!f^b(V, I^b, L^b, O^b, V')\rangle$ *in which:*

- $V$ *(resp.* $V'$*) denotes the vector of state variables before (resp. after) the execution of the behavior,*
- $I^b$, $L^b$ *and* $O^b$ *respectively denote the vectors of input variables, local variables and output variables of the operation from which b originates,*
- $StateP^b(V)$ *is the conjunction of clauses of b that do not depend from the inputs (*$I^b$*), the local variables (*$L^b$*) or the outputs (*$O^b$*),*
- $E\!f\!f^b(V, I^b, L^b, O^b, V')$, $LocP^b(V, I^b, L^b)$ *and* $InP^b(V, I^b)$ *respectively denote the effect of the behavior (assigning the after values of the state variables* $V'$ *and outputs* $O^b$*), the predicates occurring over local variables, and the predicates occurring over input variables.*

We now formally define the notion of Behavioral Model. This definition is based on a single B machine. Notice that other pre/postconditions formalisms may be mapped into this definition.

**Definition 2 (Behavioral Model).** *A Behavioral Model (BM) is a quadruplet* $BM = \langle Def, Init, Inv, Ops\rangle$ *where:*

– *$Def$ is a predicate over the static data of the model, namely the definitions of sets (types), constants and properties over the constants of the model,*
– *$Init$ is an after-predicate over the values of the state variables that gives their initial assignement,*
– *$Inv$ is the invariant predicate over the state variables of the model, it contains both typing information, and properties that have to be preserved by the operation execution,*
– *$Ops$ is set of operations in which each operation is in the* EBNF *form.*

We now illustrate the notions introduced previously. We focus on the `swap` operation of the example.

*Example 1 (EBNF of the `swap` operation).* We can give the extended definition of the `swap` operation EBNF.
$EBNF(\texttt{swap}) = swap_1[.]swap_2$ with

| | $StateP(V)$ | $InP(V, I)$ | $LocP(V, I, L)$ | $Eff(V, I, \emptyset, O, V')$ |
|---|---|---|---|---|
| $swap_1$ | $\texttt{active} \neq \emptyset \wedge$ $\texttt{ready} = \emptyset$ | *true* | *true* | $\texttt{waiting}' = \texttt{waiting} \cup \texttt{active} \wedge$ $\texttt{active}' = \emptyset$ |
| $swap_2$ | $\texttt{active} \neq \emptyset \wedge$ $\texttt{ready} \neq \emptyset$ | *true* | $\texttt{pp} \in \texttt{ready}$ | $\texttt{waiting}' = \texttt{waiting} \cup \texttt{active} \wedge$ $\texttt{active}' = \{\texttt{pp}\} \wedge$ $\texttt{ready}' = \texttt{ready} - \{pp\}$ |

in which $V = \{\texttt{active}, \texttt{ready}, \texttt{waiting}\}$, $V' = \{\texttt{active}', \texttt{ready}', \texttt{waiting}'\}$, $O = I = \emptyset$, and $L = \emptyset$ for $swap_1$ and $L = \{\texttt{pp}\}$ for $swap_2$.

## 2.3 Additional Definitions

We now define the notions of behavior enableness and crossability. These notions will be used in the next section for the computation of the transition systems.

**Definition 3 (Behavior Enableness).** *A behavior can be* enabled *if there exists a concrete state in which its precondition and the constraints applying on its parameters are satisfiable. Let $b$ be a behavior expressed in* EBNF*, $b$ can be enabled if and only if:*

$$\exists V \,.\, Def \wedge Inv(V) \wedge StateP^b(V) \wedge \exists I^b.(InP^b(V, I^b) \wedge \exists L^b.LocP^b(V, I^b, L^b)) \quad (1)$$

*is satisfiable.*

In the rest of the paper, we will denote by $enables^b(V)$ the fact that behavior $b$ can be enabled from state $V$.

**Definition 4 (Crossable behavior).** *A behavior is said to be* crossable *if its execution leads to a coherent state. More precisely, if there exists a resulting state reached by the activation of the behavior. A behavior $b$ in $EBNF$ is crossable if and only if:*

$$\exists V, I^b, L^b, O^b, V'.Def \wedge Inv(V) \wedge Beh^b(V, I^b, L^b, O^b, V') \textit{ is satisfiable.} \quad (2)$$

In the rest of the paper, we will denote by $crossable^b(V, V')$ the fact that behavior $b$ can be executed from state $V$ to reach state $V'$.

The satisfiability of the above formulas are checked using constraint solving techniques. In practice, we rely on the CLPS-BZ solver [5], a set-theoretical solver on finite domains written in SICStus Prolog. Notice that, to do so, all data domains have to be finite, namely integers are defined over a finite range of values, and sets have to be enumerated. Notice that this is not a restriction, since the B model originally aims at being used for generating test cases and thus, should not present abstract data.

We now define our technique to build symbolic transition systems, based on the behaviors that we described here.

## 3 Using Behaviors to Build the Abstraction

This section details the symbolic transition system representation. We first give the states definition and semantics, before presenting how the STS is built.

### 3.1 Symbolic Transition System

The states of our abstractions identify equivalence classes that gather the concrete states from which a given set of behaviors can be activated.

**Definition 5 (Symbolic Transition System).** *A Symbolic Transition System (STS) is defined as a quintuplet $\langle Q, q_0, \Sigma, \delta, \eta \rangle$ in which:*

- *$Q$ is a finite set of symoblic states,*
- *$q_0 \in Q$ is the inital state,*
- *$\Sigma$ is the set of the possible transitions between states, i.e. the set of behaviors extracted from the operations of the model,*
- *$\delta \in Q \times \Sigma \times Q$ is the transition relation, and*
- *$\eta$ is the characterization of a state, that is decomposed into two functions:*
  *$\eta^+ \in Q \rightarrow \mathbb{P}(\Sigma)$ which associates to a state the behaviors that can be activated at the same time, and*
  *$\eta^- \in Q \rightarrow \mathbb{P}(\Sigma)$ which associates to a state the behaviors that can not be activated at the same time.*

Notice that the $\eta^+$ (resp. $\eta^-$) function charaterizes concrete states from which the identified set of behaviors can (resp. cannot) be enabled, i.e. the conjunction of their enabling conditions (see Def. 1) is satisfiable (resp. unsatisfiable)).

**Definition 6 (State characterization predicate).** *Let $q$ be a state of a STS $\langle Q, q_0, \Sigma, \delta, \eta \rangle$, the predicate characterizing the concrete states of the model, named $\zeta(q)$ is defined by:*

$$\zeta(q)(V) = \bigwedge_{b \in \eta^+(q)} (enables^b(V)) \wedge \bigwedge_{b \in \eta^-(q)} (\neg\, enables^b(V)) \tag{3}$$

*in which $V$ is the vector of state variables.*

*Example 2 (State characterization predicates).* Consider the symbolic state $q_1$ for which $\eta^+(q_1) = \{new_1, del_1, ready_1\}$ and $\eta^-(q_1) = \{swap_1, swap_2\}$. The concrete states associated to $q_1$ satisfy the following set of constraints:

$$\exists V.\ Def \ \wedge\ Inv(V) \wedge enables^{new_1}(V) \ \wedge\ enables^{del_1}(V) \ \wedge\ enables^{ready_1}(V) \ \wedge \\ \neg enables^{swap_1}(V) \ \wedge\ \neg enables^{swap_2}(V)$$

which corresponds to:

$$\begin{aligned}\exists(a,r,w).\ Def \wedge Inv(a,r,w) \wedge\ & \exists pp^{new_1}.(pp^{new_1} \in PID \ \wedge\ pp^{new_1} \notin a \cup r \cup w) \wedge \\ & \exists pp^{del_1}.(pp^{del_1} \in w) \ \wedge \\ & a = \emptyset \ \wedge \exists pp^{ready_1}.(pp^{ready_1} \in w) \ \wedge \\ & \neg(a \neq \emptyset \ \wedge\ r = \emptyset) \ \wedge \\ & \neg(a \neq \emptyset \ \wedge\ r \neq \emptyset \wedge \exists pp^{swap_2}.pp^{swap_2} \in r)\end{aligned}$$

in which $a$, $r$ and $w$ respectively denote state variables `active`, `ready` and `waiting`. This constraint satisfaction problem reduces to the following:

$$\texttt{active} = \emptyset \wedge \texttt{ready} = \emptyset \wedge \texttt{waiting} \neq \emptyset \wedge \texttt{waiting} \subset \texttt{PID}$$

which has 42 solutions, one of them being $\texttt{active}{=}\texttt{ready}{=}\emptyset$, $\texttt{waiting} = \{\text{p1}\}$.

The existence of concrete states is decided using constraint solving techniques, which aim at finding an instanciation for the state variables and local behavior parameters.

## 3.2 Building the Symbolic Transition System Using CLP

Our approach for computing the STS is based on the approach originally proposed by Bert and Cave in [3] and implemented in the GeneSyst tool [9]. This technique consists in two steps: first, the identification of the states, and, second, the computation of the transitions between the states. Contrary to the GeneSyst tool, which relies on theorem proving, our approach uses constraint solving techniques to compute the feasibility of transitions.

**Step 1. State partitioning.** Let $\Sigma^V$ be the set of behaviors whose enableness condition depends on the state variables of $V$. The complementary behaviors represent behaviors whose enabling does not depend on any state variables, i.e. behaviors that can be activated from any state of the system, and thus, that are not discriminating.

Each symbolic state is defined by $\eta$ defined so as to satisfy the following set of properties:

($a$) if an operation has no behavior in $\Sigma^V$ the operation is not considered
($b$) at most one behavior for each operation is in $\eta^+$
($c$) if an operation has no behavior in $\eta^+$, all its behaviors appear in $\eta^-$.

**Proposition 1 (Partitioning of the model state space).** *This symbolic state definition represents a complete partition of the model state space.*

*Proof (Sketch of proof of Proposition 1).* This proof can be done by contradiction. Assume that our definition does not represent a complete partition of the model state space. Then, there are two cases: either $(i)$ there exists at least a state that is not in the partition, or $(ii)$ there exists at least a state that may belong to two symbolic states. Case $(i)$ is caught by properties $(b)$ and $(c)$, which aim at representing the possible combinations of behaviors that can be enabled for an operation: either a single behavior, or no behavior at all. The exhaustiveness of the combinations guarantees that all cases are considered. Case $(ii)$ is caught by property $(a)$, which prevents two concrete states from enabling different behaviors that do not depend on the state in which they are executed.

Thus, the theoretical number of states is the product $\Pi(|\Sigma_{op} + 1|)$ in which $\Sigma_{op}$ designates the behaviors extracted from operation $op$. The additional +1 takes into account the fact that no behavior from a given operation can be enabled. In practice, this number may also be reduced in presence of contradictory predicates between the combined behaviors.

*Example 3 (Scheduler states partitioning).* The scheduler example contains 4 operations, and a total of 6 (state-variable dependent) behaviors: 1 for `new` and `del`, and 2 for `ready` and `swap`. The `observe` operation has 3 behaviors that can be enabled from any state. Since none of its behaviors depend on the state variables, this operation is thus not considered as characteristic for the definition of the symbolic states, and is not taken into account.

The following table summarizes the possible combinations that can be produced, for each operation. In this table, $op_\emptyset$ represents the fact that no behavior from operation $op$ will be activated (i.e. all the behaviors of $op$ will be in $\eta^-$).

| Operation | `new` | `del` | `ready` | `swap` |
|---|---|---|---|---|
| Behaviors | $new_\emptyset$, $new_1$ | $del_\emptyset$, $del_1$ | $ready_\emptyset$, $ready_1$, $ready_2$ | $swap_\emptyset$, $swap_1$, $swap_2$ |

In theory, the maximal number of symbolic states is $2 \times 2 \times 3 \times 3 = 36$. In practice, only 10 combinations produce satisfiable conjunctions of behavior enabling conditions (e.g. $ready_1$ and $swap_1$ respectively require `active` $= \emptyset$ and `active` $\neq \emptyset$, which is inconsistent, removing all combinations including these two behaviors from the set of symbolic states).

Notice that the (concrete) initial state of the original model is mapped to one of these identified states. On the example, $\eta^+(q_0) = \{new_1\}$ and $\eta^-(q_0) = \{del_1, ready_1, ready_2, swap_1, swap_2\}$.

**Step 2. Transitions computation.** After having identified the states of our STS, we then compute the transitions between them. As explained previously, the transitions are the behaviors that can be activated between the states. Thus, for each pair of states, we compute the crossability of a behavior, from a first state, in order to reach the second one. The worst case number of transitions is thus $|Q|^2 \times |\Sigma|$.

**Definition 7 (Feasible transition between states).** *Let $q$ and $q'$ be two states from an STS, and let $b$ be a behavior. The transition from $q$ and reaching $q'$ by $b$ is feasible if and only if:*

$$\exists V, V' \;.\; \zeta(q)(V) \wedge crossable^{b}(V, V') \wedge \zeta(q')(V') \quad \textit{is satisfiable.} \tag{4}$$

Again, we consider constraint solving techniques to check the satisfiability of the above predicate. Let us now illustrate the computation of a transition on an example.

*Example 4 (Transition computation).* Suppose we want to check the feasibility of $new_1$ between the initial state $q_0$, defined by $\eta^+(q_0) = \{new_1\}$ and $\eta^-(q_0) = \{del_1, ready_1, ready_2, swap_1, swap_2\}$, and $q_1$, defined by $\eta^+(q_1) = \{new_1, del_1, ready_1\}$ and $\eta^-(q_1) = \{swap_1, swap_2\}$.

The existence of the transition $(q_0, new_1, q_1)$ is determined by the satisfiability of the formula:

$$\begin{aligned}\exists a, r, w, a', r', w' \;.\; & \neg(\exists pp^{del_1}.pp^{del_1} \in w) \;\wedge \ldots \wedge \\ & \exists pp^{new_1}.(pp^{new_1} \in PID \;\wedge\; pp^{new_1} \in a \cup r \cup w \;\wedge \\ & \qquad w' = w \cup \{pp^{new_1}\} \;\wedge\; a' = a \;\wedge\; r' = r) \;\wedge \\ & w' \neq \emptyset \;\wedge\; w' \subset PID \;\wedge\; a' = \emptyset \;\wedge\; r' = \emptyset\end{aligned}$$

in which $a$, $r$, and $w$ respectively denote `active`, `ready` and `waiting`. This reduces to the following set of constraints:

$$\begin{array}{ll} \texttt{active} = \texttt{active}' = \texttt{ready} = \texttt{ready}' = \emptyset & \text{from } \zeta(q_1) \\ \texttt{waiting} = \emptyset & \text{from } \zeta(q_0) \\ \texttt{waiting}' \neq \emptyset \;\wedge\; \texttt{waiting}' \subset \texttt{PID} & \text{from } \zeta(q_1) \end{array}$$

which admits 5 solutions, corresponding to the initial state of the model (`active` = `ready` = `waiting` = $\emptyset$) and the "first" reachable state of the model (`active` = `ready` = $\emptyset$ and `waiting` = $\{pp\}$, with $pp \in \{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}, \texttt{p5}\}$.

*Example 5 (Symbolic Transition System of the Scheduler).* The symbolic transition system computed from the Scheduler example is depicted in Fig. 2. This STS contains 10 states, that are characterized by the set of behaviors that can be enabled from them. State $q_0$ represents the initial state of the STS. These 10 states are related by 40 transitions. For readability, the `observe` operation behaviors have not been represented.

Notice that the number of transitions is significantly lower that its worst case value ($10^2 \times 6 = 600$ transitions).

As explained before, the computation of the symbolic states and the transitions between them is done using constraint solving techniques. Notice that proof techniques may also be used. Nevertheless, theorem proving relies on the underlying theories used to express the formula, that need to be decidable to avoid proof defaults. Moreover, the expressiveness of the B notation (involving relations, functions, etc.) may not be formalized into decidable theories. On the opposite,

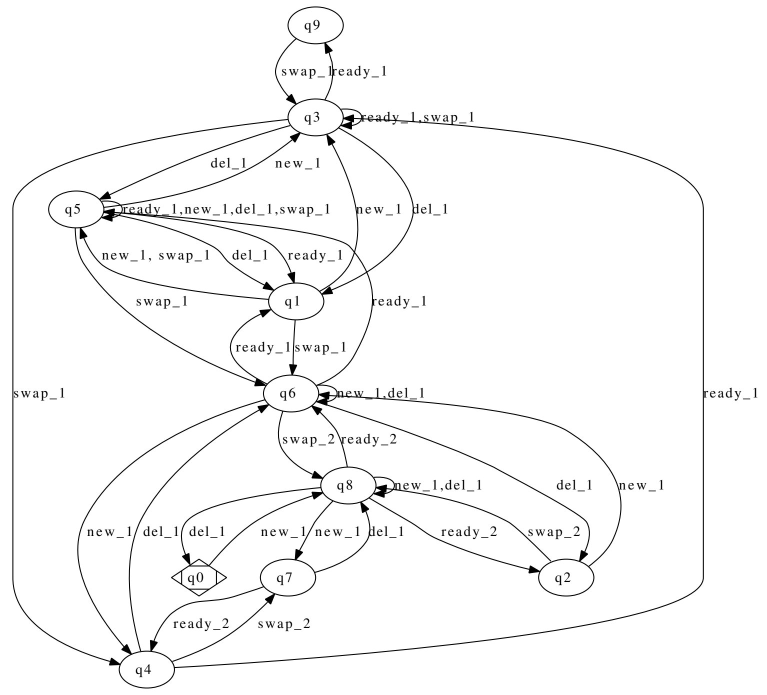


**Fig. 2.** Symbolic Transition System of the Scheduler example

constraint solving techniques guarantee the termination of the computation (i.e. in the worst case, all possible solutions of the Constraint Satisfaction Problem are enumerated before concluding to unsatisfiability of the formula). Thus, we choose to use a customized set-theoretical solver, namely CLPS [5].

In practice, the resulting graph may be non-deterministic (as illustrated in Fig 2). This specificity has to be taken into account in the test generation approach that we consider. It is now described.

## 4 Test Generation from the Abstraction

We present in this section how to generate test cases from the abstraction.

As explained in the previous section, the resulting abstraction is *possibly* non-deterministic, meaning that a same transition may reach more than one state. Thus, it would make no sense to generate the test cases *offline* (i.e. by considering only the STS) since the larger majority of the sequence would not be executable. To tackle this potential problem, we propose to perform the test generation in an *online* way: while the STS is explored, the original B model is animated. At each step, an identification of the current STS state is performed so as to provide a feedback on the STS exploration.

**Algorithm 1:** Test generation algorithm for a targeted transition

```
Data: model $\mathcal{M}$, STS $\mathcal{S} = \langle Q, q_0, \Sigma, \delta, \eta \rangle$
Result: testSuite
begin
    testSuite ← $\emptyset$ ;
    foreach $(q_f, b, q_t) \in \delta$ do
        test ← [] ;
        pathsToTarget ← findPath($q_0$,$q_f$) ;
        foreach path ∈ pathsToTarget do
            curState$_\mathcal{M}$ ← initialize($\mathcal{M}$) ;
            foreach $(q_{fp}, b_p, q_{tp}) \in$ path do
                curStateAfter$_\mathcal{M}$ ← executeBehavior($\mathcal{M}$,curState$_\mathcal{M}$,$b_p$) ;
                if checkState($\zeta(b_t)$,curStateAfter$_\mathcal{M}$) then
                    curState$_\mathcal{M}$ ← curStateAfter$_\mathcal{M}$;
                    test ← test ˆ $(q_{fp}, b_p, q_{tp})$;
                else
                    break and backtrack to next path;
            curState$_\mathcal{M}$ ← executeBehavior($\mathcal{M}$,curState$_\mathcal{M}$,$b$) ;
            checkState($\zeta(q_t)$,curState$_\mathcal{M}$) ;
            test ← test ˆ $(q_f, b, q_t)$ ;
            testSuite ← testSuite ∪ {test} ;
            break ;
```

Our test selection criterion aims at the coverage of all transitions $q \xrightarrow{b} q'$ for each $b \in \Sigma$. Our objective is to introduce the most possible variety in the preambles that reach the targeted states. Knowing the topology of the model, thanks to the abstraction, we can now use it to compute the paths to the target. For each transition, the idea is to cover a path leading to it from the initial state.

Algorithm 1 summarizes the test generation algorithm, and especially the replay of a path from a given path. The basic idea is to start from a given path and monitor its symbolic execution of the original B model, at each step. Its aim is to generate a test suite which consists of one test for each transition in the graph of the abstraction. For each transition, a test is built as follows:

– The algorithm first finds the paths from the initial state $q_0$ of the abstraction to the source state of the considered transition $q_f$, computed by function *findPath*. This function uses a Dijkstra-like implementation of the path computation, done in polynomial time. It represents a choice-point in the sense that it is able to iterate over the possible paths that lead to a specific state, by first computing loop-free paths, in which loops are unfolded if this latter can not be replayed on the model. The variable *curState* is initialized with the values of variables of the model at initial state.
– For each path, the algorithm consider the successive transitions in it, and executes the associated behavior on the model (function *executeBehavior*).

It then checks that the predicate associated to the target state (using the $\zeta$ function) is satisfied on the resulting model state.
- If the check succeeds, the current transition of the path is concatened (denoted by the ˆ operator) to the test, and the algorithm continues with the rest of the path.
- If not, the execution backtracks to the next path (iterating the possible loops over the current path).
- When the path has been played entirely, the test is concluded with the targeted transition. Then, the algorithm continues with the next test.

The complexity of this algorithm is related to the worst case complexity of the *findPath* function. In practice, this function is bounded in the number of steps done for unrolling loops. Its worst case is thus exponential. Nevertheless, the heuristics consisting in unrolling the loops of the path makes it possible, in practice, to have acceptable computation times.

This algorithm has been implemented in SICStus Prolog, and communicates with the CLPS solver and the engine for animating the B model. The implementation uses backtracking mechanisms to iterate over the possible paths that lead to a given state. Backtracking also avoids the re-computation of the complete sequences composing the different paths by factorizing the evaluation of sequences prefixes.

## 5 Experimentation

We now report on the experimentation we have designed. The goal of the experimentation is to show the usefulness of the abstraction we propose for test generation, by comparing its results with a commercial test generation tool, named Leirios Test Generator [12]. This tool is also based on the coverage of the behaviors, but its test generation strategy only considers one path for each behavior, thus producing short test cases. We show here that the abstraction is helpful to compute a wider variety of test cases which improve the test detection capabilities of the tests.

### 5.1 LTG Test Genereation Strategy

The test generation strategy of LTG is based on the decomposition of the B operations into behaviors. Contrary to our approach, the LTG test generation algorithm does not take into account the topology of the model states.

For each behavior, a test target corresponds to its enableness condition, as defined in Def. 1. The test targets represent a set of concrete states from which it is possible to activate the considered behavior. The aim of LTG is to produce tests that will cover the behaviors of the system.

The computation of the *preamble* (the sequence of operations that reaches the target) is done automatically by LTG. To achieve that, the tool uses a customized state exploration algorithm that explores on-the-fly the symbolic states of the model until reaching the specified target. In most of the cases, the paths found by the exploration algorithm turn out to be the shortest paths reaching the target.

### 5.2 Comparison with LTG Using Mutational Analysis

The application of the abstraction-based test generation algorithm on the example produced 40 tests, one per transition, of average length 6.25 steps (vs. 3 for the 6 tests produced by LTG).

In order to check the improvement brought by our technique, we designed a mutational analysis on the Scheduler example. We thus produced manually 23 mutants of the original specification and we ran the tests produced using our technique and compared with the executions of the tests produced by LTG. The mutations were done on the model itself. They consisted in modifications/deletions of conditions in the decisions of IF substitutions, and addition/modifications/deletions of assignments. We replayed the tests on the model, and checked the feasibility of the sequences. A mutant is killed by a test if the test can not be replayed on it (trace equivalence). In general, the test execution time is negligible (less than one second using our dedicated animation engine).

LTG tests achieve a mutant detection score of 13/23, whereas our technique scores 18/23. Interesting points are that (*i*) our approach detects the same mutants as LTG (i.e. there is no mutant that is killed by LTG and not by our approach), and (*ii*) the mutants that are still alive with our tests also remain alive with LTG. As expected, our test generation technique improves the test generation approach of LTG.

*A word on scalability.* We conducted another experiment, with a large size model of an electronic purse. Although the model admitted more than $10^{12}$ concrete states, and 20 behaviors (among 5 operations), the abstraction, containing 23 states and 303 transitions was computed in an acceptable time of 1min 42sec. The test generation algorithm took less than one minute to generate more than 100 test cases (mainly due to the fact that no unrolling of loops was necessary).

## 6 Related Works

We compare here our approach with other abstraction computation techniques and/or test generation approach from behavioral models.

The method implemented in the Agatha tool [18] also computes an abstraction from a model, but by applying a symbolic execution technique. The technique consists of an exploration of the model from the initial state by using an inclusion criterion on explored symbolic states. The tool computes a tree, in which each leaf is a symoblic state that is a subset of another state in the tree. Then, the tests are computed by exploring this tree. Other approaches rely on a first computation of an abstraction that is then refined using counter-examples [8] that are traces that should not be accepted by the abstraction. In SpecExplorer [21], a model program is used to describe the model that will be explored using symbolic techniques and model-checking to produce test cases. The methods presented in [6] and implemented in the STG tool [13] use an abstraction defined by the user and modelled by an IOSTS. These approaches use *test purposes* synchronized with the abstractions, both defined as IOSTS. Then the synchronized product is used to generate test cases after an optimization step, which consist of pruning the unreachable states by abstract interpretation.

In the domain of test generation from B machines, the ProTest test generator [19] generates test cases from an exploration of a labeled transition system built using the animation of a B machine, using the ProB animator. The major difference with our work is that ProB enumerates all the accessible states of the B model, which is untractable on large systems. Even if state reduction heuristics have been implemented into ProB [17], it seems, to the best of our knowledge, that the resulting transition system is not used for test generation, but rather for property verification.

Contrary to these approaches, our technique does not perform an exploration of the model execution in order to build the symbolic states. Assuming that the invariant of the B model captures all the reachable states of the model, the partitioning that we propose can possibly identify states that would not have been reached when exploring the model if using an inappropriate symbolic state mergin criterion. In return, our abstraction is not deterministic and the test generation technique that we employ has to be able to deal with it. Moreover, the above-mentionned approaches consider relatively simple data types, such as integers and booleans, whereas we are able to reason on complex data structures such as sets and functions/relations.

A similar state partitioning technique is presented in [10]. The authors propose the operation in disjoint Z schemas for each operation (accordingly to the DNF decomposition initially proposed by Dick and Faivre [11]), and also provide a partition of state variables values. Our decomposition goes one step further by considering combinations of behaviors to provide the state partitioning, resulting in a more precise abstraction. This kind of partitioning represents the main originality of our approach.

The closest work to ours is reported in [4]. In this work, the GeneSyst tool [9] is used to compute a symbolic transition system based on a syntactic abstraction of a B model. Unfortunately, this work suffers from several limitations. First, the state partitioning is done by focusing on a given subset of the state variables whose domains are partitioned in a systematic way, which may cause subsequent transitions to be unactivable. Second, the GeneSyst tool is based on a prover which may fail to deduce the validity of a formula, leaving proof obligations unsolved. Our approach using constraint solving techniques avoids this problem. Finally, the computation of the tests does not take into account that a large majority of traces produced from the exploration of the abstraction can not be replayed on the original model. Our approach tackles this problem by considering the animation of the original model and the exploration of the graph altogether.

## 7 Conclusion and Future Works

We have presented in this paper a technique to build an abstraction from a B model. The states of the abstraction are characterized by a set of behaviors (extracted from the B model operations) that can be activated at a same time.

The transitions between the states are the behaviors of the model. Their existence is computed using constraint solving techniques. This abstraction makes it possible to provide an approximate reachability analysis that can be used to help the design of model-based test cases. We have then proposed to exploit the knowledge of the model topology to improve the variety of the test cases that can be produced by considering the activation of all the transitions of the abstraction, thus covering various enabling states for a given behavior.

Since the resulting abstraction may be non-deterministic, there is no guarantee that the potential paths computed from it can be instantiated when animating the model. To address this problem, we couple the exploration of the STS with the animation of the model. The experiments have shown that most of the paths could be instantiated, modulo the unfolding of intermediate loops in the abstraction.

We are now looking for a way to remove (totally or partially) the non-determinism that is induced by our approach. Our current work considers an extended definition of the symbolic states, by refining them with a set of discriminating behavior sequences that can (or can not) be activated from them. In addition, we plan to apply our technique on larger case studies, since the first results on realistic models are promising. Finally, we plan to study the influence of the test generation algorithm on the resulting tests.

**Acknowledgments.** The authors would like to thank the reviewers of the TAP'2010 conference for their useful comments and remarks on this paper.

## References

1. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, New York (1995)
3. Bert, D., Cave, F.: Construction of finite labelled transistion systems from b abstract systems. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 235–254. Springer, Heidelberg (2000)
4. Bouquet, F., Bué, P.-C., Julliand, J., Masson, P.-A.: Test generation based on abstraction and test purposes to complement structural tests. In: A-MOST 2010, 6th Int. Workshop on Advances in Model Based Testing, in Conjunction with ICST 2010, Paris, France (April 2010)
5. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B: A constraint solver to animate a B specification. International Journal on Software Tools for Technology Transfer, STTT 6(2), 143–157 (2004)
6. Calame, J.R., Ioustinova, N., van de Pol, J., Sidorova, N.: Data abstraction and constraint solving for conformance testing. In: 12th Asia-Pacific Software Engineering Conference (APSEC 2005), pp. 541–548. IEEE Computer Society, Los Alamitos (2005)
7. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: Stg: a tool for generating symbolic test programs and oracles from operational specifications. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference, pp. 301–302. ACM, New York (2001)

8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Allen Emerson, E., Prasad Sistla, A. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
9. Potet, M.-L., Bert, D., Stouls, N.: GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties. In: Treharne, H., King, S., Henson, M. C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 299–318. Springer, Heidelberg (2005)
10. Derrick, J., Boiten, E.A.: Testing refinements of state-based formal specifications. Softw. Test., Verif. Reliab. 9(1), 27–50 (1999)
11. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Larsen, P.G., Woodcock, J.C.P. (eds.) FME 1993. LNCS, vol. 670, pp. 268–284. Springer, Heidelberg (1993)
12. Jaffuel, E., Legeard, B.: LEIRIOS Test Generator: Automated Test Generation from B Models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 277–281. Springer, Heidelberg (2006)
13. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 349–364. Springer, Heidelberg (2005)
14. Jones, C.B.: Systematic software development using VDM, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1990)
15. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE, 1090–1123 (1996)
16. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
17. Leuschel, M., Turner, E.: Visualising Larger State Spaces in ProB. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 6–23. Springer, Heidelberg (2005)
18. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.-P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA 2003, Automated Technology for Verification and Analysis (2003)
19. Satpathy, M., Leuschel, M., Butler, M.: ProTest: An Automatic Test Environment for B Specifications. Electronic Notes Theoretical Computer Science (111), 113–136 (2005)
20. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems 29(1), 49–79 (1996)
21. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008)

# Author Index