AFM'08: Third Workshop on Automated Formal Methods 14 July 2008 Princeton, New Jersey

John Rushby and Natarajan Shankar (Editors)

SRI International
Computer Science Laboratory
Menlo Park CA 94025 USA
{rushby | shankar}@csl.sri.com

Table of Contents

Preface

This volume contains the proceedings of the Third Workshop on Automated Formal Methods held on 14 July 2008 in Princeton, New Jersey in association with the Conference on Automated Verification (CAV). The first AFM workshop was held as part of the Federated Logic Conference in July 2006 in Seattle, Washington, and the second in association with the Automated Software Engineering (ASE) conference in November 2007 in Atlanta, Georgia. The focus of the AFM workshop is on topics related to the SRI suite of formal methods tools including PVS, SAL, and Yices. We received 8 submissions of which 6 were accepted for presentation at the workshop.

In addition to the contributed papers, the conference included an invited paper by Grégoire Hamon on *Simulink Design Verifier–Applying Automated Formal Methods to Simulink and Stateflow*, short tutorials on each of Yices, SAL, and PVS, and a session of short presentations on current research and discussions of extensions and enhancements to these verification tools.

We thank the distinguished members of the program committee as well as the external referees for their thorough and thoughtful reviews of the submitted papers. The paper submission and reviewing process was managed through the Easychair conference management system. We also thank the organizers associated with CAV 2008, particularly the program co-chair Aarti Gupta, and the workshops chair Byron Cook. We also received help from the ACM Publications Coordinator Adrienne Griscti and our SRI colleagues Bruno Dutertre, Sam Owre, and Ashish Tiwari.

We hope the AFM workshop series will continue to serve as a forum for communication and cooperation between the developers and users of automated formal verification tools.

John Rushby
Natarajan Shankar
Menlo Park, California

Programme Chairs

John Rushby
Natarajan Shankar

Programme Committee

Aaron Bradley	University of Colorado, Boulder
Supratik Chakraborty	IIT Mumbai
Bernd Finkbeiner	Universitat des Saarlandes
Jean-Christophe Filliatre	Universite Paris Sud
Marcelo Frias	Universidad de Buenos Aires
Chris George	United Nations University IIST
Mike Gordon	Cambridge University
Constance Heitmeyer	Naval Research Laboratory
Paul Jackson	Edinburgh University
Steve Johnson	Indiana University
Leonardo de Moura	Microsoft Research
Pete Manolios	Northeastern University
David Monniaux	Verimag
Cesar Munoz	National Institute of Aerospace
David Naumann	Stevens Institute of Technology
Kazuhiro Ogata	JAIST
Corina Pasareanu	NASA Ames
Lee Pike	Galois Inc
Koushik Sen	University of California, Berkeley
Maria Sorea	EADS
Ofer Strichman	The Technion
Mark Utting	Waikato University
Michael Whalen	Rockwell Collins Inc

External Reviewers

Claude Marché

Simulink Design Verifier — Applying Automated Formal Methods to Simulink and Stateflow

Grégoire Hamon
The MathWorks
Natick, MA – USA
Gregoire.Hamon@MathWorks.com

ABSTRACT

We present Simulink Design Verifier, a tool distributed by The MathWorks to help Simulink users in their verification and validation activities. Simulink Design Verifier automatically generates test input sequences for a model or proves properties about this model. The tool is based on automatic theorem proving and model-checking techniques and aims at a high level of integration with Simulink to make formal verification techniques easier to incorporate into engineers' workflows. In this presentation, we describe the tool, highlight some of the technical choices that were made during its conception, and discuss user experiences with it.

1. INTRODUCTION

The advent of model-based design environments in the design of control systems has brought an interesting step between the specification and the code — the model. The model can be seen as an executable specification of the system under design. From a formal methods point of view, it brings an opportunity to apply automatic tools much earlier in the development cycle, and at a higher level of abstraction. Simulink [2] is a very widely used model-based design environment. It supports the description of both discrete and continuous time models in a graphical block-diagram language. Models can also include state-machine diagrams using Stateflow, or imperative code written in Embedded Matlab. The environment has extensive capabilities for simulation and code-generation.

Simulink Design Verifier [3] is a recently introduced tool that provides automatic test case generation and property proving for models. It makes use of automatic theorem-proving and model-checking technologies and fully integrates into Simulink. The tool was conceived to fit into users' workflow if needed, requiring as little interaction as possible for simple activities. It is also extensible so that advanced users that want to specialize the tool to their needs can do so.

In this presentation, we will first present Simulink Design Verifier, then highlight some of the design decisions that were made during its conception, as well as discuss user experience with the tool and directions for improvement.

2. ANALYZING SIMULINK

Simulink Design Verifier has two distinct uses: test case generation on one hand, property proving on the other hand. These two activities make use of the same core technology, based on model-checking. As was previously reported [1, 4], a model-checker can be efficiently used as an automatic test case generator, as long as the model-checker can generate concrete counter-examples when a property is violated. The engine used by Simulink Design Verifier is provided by Prover Technology, it is based on SAT-solving techniques and can decide problems expressed in a range of theories including linear integer and rational arithmetics.

The architecture of the tool is fairly straightforward: the Simulink model to analyze is translated to the input language of the model-checker; during the translation, properties and test objectives are inserted. Two related questions when considering the analysis of Simulink models are the subset of the language that should be supported and how to do the translation.

The tool is restricted to the analysis of discrete controller models. Simulink can describe a much wider range of systems, but with today's verification technology, and when designing a general purpose tool, this is not really a choice. This is also the subset of Simulink that is supported by the code-generator. The only other limitations are at the block level, for example trigonometric functions are not directly supported. Both Stateflow and Embedded Matlab are supported, and all data types including fixed-point arithmetics are supported. The translation itself uses the code-generator front-end to get an abstract representation of the model that is then translated to the input language of the model checker.

3. PROPERTIES AND TEST OBJECTIVES

Once the model is translated, we need to define objectives for the analysis. In the simplest cases, the objectives are inferred from the model itself. This is mainly for users interested in generating test cases for common coverage criteria. Simulink also has assertion blocks that can stop simulation if a particular condition is reached; if a model contains such blocks, the tool can try proving that these assertions cannot be raised.

More advanced users want to define their own objectives — objectives corresponding to functional testing requirements, or complex properties. Design Verifier comes with a library of blocks that can be used to define objectives as observers: Simulink itself is used as the specification language for the property. The library itself only contains basic blocks comparing a signal to a value, or a set of values. This library has equivalent constructions in Stateflow. Objectives can be defined inside the model under analysis itself, or outside using Simulink modular capabilities. Currently, the tool only supports the definition of safety properties.

Using Simulink as the specification language is for a large number of users a very good choice: they already know the language, it is expressive enough to describe complex objectives. Some users however would need to express their objectives at a higher level, closer to the requirements. The links between requirements and model is an open area for research. Simulink provides a tool to link elements of a model to an informal requirement, this can be used to link a requirement to an objective written in Simulink.

4. UNDERSTANDING THE RESULTS

Three artifacts are generated by the Simulink Design Verifier: a data file, a test harness, and a report. The data file contains all the information that might be needed about the analysis that was run, including the model version number, the configuration of the tool, the details of any test case or counter-example that was found. The test harness is a new model, containing both the model under analysis and an input generator that can drive the model using the test cases and counter-examples that were generated. The report presents in an easy to read way details about the results of the analysis.

These aspects are often considered side-issues to the formal analysis, but from a user perspective, they are critical. The amount of data produced when generating test cases for coverage can be huge, and the user needs to be able to reuse, inspect, and understand these results. When proving properties, the results of an analysis can be greatly confusing — for example if a user gets a counter-example for a property that he thought was valid, he needs to be able to review precisely what happened to understand if the property itself was expressed correctly, if external constraints on the environment were correctly set, etc.

5. DISCUSSION

Simulink Design Verifier is a new tool, users are still learning how to best use it, and incorporate it in their design process. Initial feedback is very positive, in general there is an important and growing interest in formal verification.

The first question that comes when talking about model-checking techniques is often "what about scalability?" — scalability is of course a problem, and will probably always be, but is not the one big problem. Often, as users look at bigger and bigger models, other problems come before scalability — compatibility problems in particular. Scalability also depends on the activity: in general test case generation can handle larger models. Some scalability problems are "contained" such as the known counter problems: a system has to exhaust a (big) counter to continue execution — such cases can be addressed by special treatment; Simulink Design Verifier has abstractions for some counter patterns.

Compatibility problems often arise when users start looking at bigger models - these are either coming from user-defined blocks for which Simulink Design Verifier has no information, or from non-linear operations. The tool has a mechanism for users to give a Simulink approximation of their user-defined blocks, and automatically use this representation. Non-linear operations are a big problem, and the main limitation of the technology used.

Although the same core technology is used to do both property proving and test case generation, the technology is used very differently. Test case generation, at least for coverage criteria, involve a huge number of objectives, most of which are actually easy to reach, and spawning over the complete model. Property proving often involves a much smaller number of objectives, but they can be very hard to reach. Another key difference is that the validity of a test can be checked by simulation, opening the way to aggressive simplifications and abstractions. On the other hand, test case generation relies on the ability to generate a concrete counter-example, which can make some abstractions applicable only to property proving.

Using model-checkers to generate test cases is in practice very appealing. Very short test case can be discovered if that's the need, but the analysis can also be done so that it will find longer ones. The ability to get the result that there is no test case for a configuration, in truth proving a property rather than disproving it, is very important as it definitely indicates a problem in the model.

Open areas of investigation are plentiful. One of them is understanding how such tools can better integrate in user workflow, and this leads to understanding how to help users in expressing high-level requirements and bringing them to the tool for analysis.

6. REFERENCES

[1] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[2] The MathWorks, http://www.mathworks.com/products/simulink/. *Simulink*, 2008.

[3] The MathWorks, http://www.mathworks.com/products/sldesignverifier/. *Simulink Design Verifier*, 1.2 edition, 2008.

[4] S. Rayadurgam and M. P. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03*, Greenbelt, Maryland, December 2003.

Using Yices as an automated solver in Isabelle/HOL

Levent Erkök John Matthews

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97004
{levent.erkok,matthews}@galois.com

ABSTRACT

We describe our integration of the Yices SMT solver into the Isabelle theorem prover. This integration allows users to take advantage of the powerful SMT solving techniques within the interactive theorem proving environment of Isabelle, considerably increasing the automation level for a significant subset of Isabelle/HOL.

1. INTRODUCTION

This paper describes the Isabelle `ismt` tactic,[1] developed by Galois to seamlessly integrate the Yices SMT solver within the interactive theorem proving environment of Isabelle, thus increasing the automation level considerably as well as providing counterexample information back to the user when Yices detects a formula is invalid; similar to PVS's `yices` strategy [6].

The `ismt` tactic is freely available on the internet with a permissive BSD-style license [5].

1.1 Yices

Yices is a modern SMT solver that supports uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit vectors, λ-expressions, and quantifiers [7, 11, 16]. Yices's input language is based on a LISP like syntax extended with type declarations. Nevertheless, Yices's input language is still significantly more restrictive than Isabelle/HOL [25]. For instance, Yices currently does not support parameterized datatype declarations, mutual or nested recursion in datatypes, or bounded quantification over sets. Most importantly, Isabelle/HOL's and Yices's type systems are substantially different: While the former has a polymorphic type system, Yices only allows monomorphic definitions with uninterpreted types. However, we still consider Yices a suitable target for integration, and our tactic is able to translate many of Isabelle/HOL's extra features into Yices equivalents.

Most SMT solvers support the *SMT-Lib* language [9], which is the input format used in the SMT competitions. However, we chose to avoid translation through this medium for two reasons. First, SMT-Lib only supports a limited set of theories [8] which does not include datatypes, tuples, records, etc. Second, the SMT-Lib language is not incremental. That is, facts cannot be asserted or retracted in the middle of a proof. While we do not currently make any significant use of Yices' native incremental API, our future plans do include taking full advantage of this functionality.

1.2 Modes of integration

The `ismt` tactic invokes Yices as an *external oracle*, meaning that it trusts the soundness of Yices and our translator. Whenever Isabelle produces a theorem via an external oracle, it attaches a *trust tag* to it, as well as to any other theorem that uses this theorem in its proof. The trust tag says which tool was invoked, and the formula the tool proved. Isabelle displays a "`[!]`" annotation on any theorem containing trust tags, and the list of trust tags associated with a theorem can also be directly queried.

Assuming Yices supports a mode in the future where explicit proof objects are returned, then we would also like to build a *proof replay* mode for `ismt` where the proof object is used to reconstruct a purely Isabelle proof of the theorem that would not contain any trust tags.

2. RELATED WORK

There have been several attempts at integrating SMT solvers into theorem proving environments. In this section we review the most relevant ones to our work.

2.1 Integration of Yices with PVS

The Yices SMT solver can be used as an end-game solver in PVS [6]. The translation from PVS to Yices is much more direct than ours, since Yices and PVS share the same type system. Similar to our work, Yices acts as a trusted solver in PVS, whose results are not verified independently. Unlike our tactic, however, the models generated by Yices are not translated back to PVS notation: A failed proof attempt by Yices is simply interpreted by PVS as a skip, having no effect on the proof state.

2.2 The smt tactic

Barsotti et al. describes how to integrate generic SMT solvers with Isabelle [10, 14]. Similar to our work, their tac-

[1]The name `ismt` was chosen to avoid conflict with an already existing `smt` tactic (see Section 2.2), and also to emphasize our future plans of taking advantage of incrementality.

AFM08: Automated Formal Methods '08 Princeton, New Jersey, USA

tic works as an oracle as well, i.e., no proof reconstruction is done. Unlike our work, however, they target the SMT-Lib language [9] as their translation medium, allowing them to use arbitrary SMT solvers that support this common language. Due to the limitations of SMT-Lib, however, Barsotti et al. forgo expressive power, not being able to support datatypes, case-statements, λ-expressions, tuples, records, etc., which are the basic pillars of the Isabelle/HOL language. We consider the expressive power afforded by the richer internal language of individual SMT solvers well worth the cost of building custom translators.

2.3 The rv tactic

Fontaine et al. describe how to perform proof reconstruction using proof generating SMT solvers [18]. In their work, they use haRVey [3] to generate "proof hints," that can later be replayed by Isabelle via the `rv` tactic. While this is precisely the technique we would like to use in the future, currently haRVey only supports propositional logic, uninterpreted functions, and linear arithmetic; a rather limited language compared to what our tactic can handle.

2.4 Integration of CVC-Lite with HOL-Light

The final related work we would like to review is the integration of CVC-lite SMT solver [1] within HOL-Light [4], as described by McLaughlin, Barrett, and Ge [24]. In this work, proofs generated by CVC-Lite are represented as tree-like data structures [13], which are parsed back and replayed in the HOL-Light theorem prover. This translation of proof trees is especially made easy since the CVC-Lite logic is very close to a subset of the HOL-Light logic.

McLaughlin et al. point out that to integrate the CVC-Lite proofs about arrays, for instance, they had to extend HOL-Light to understand CVC-Lite inference rules. While they note that this was a trivial task in this particular case, it is not clear how easy it would be to repeat this exercise to other theories of interest.[2]

2.5 Integration of UCLID with ACL2

Manolios and Srinivasan [23] modified the ACL2 theorem prover to provide an oracle-like facility for solving ACL2 subgoals using the UCLID [22] SMT solver. To prove an ACL2 goal P correct, they first use an untrusted translation from ACL2 to UCLID, obtaining a UCLID formula P'. If UCLID proves P' valid, then a trusted translator from UCLID to ACL2 is applied, obtaining an equivalent ACL2 assertion G. Finally, ACL2 itself is used to prove that G implies P. If this final proof goes through, then they assert that the original goal P must be a theorem, modulo the soundness of the translation from UCLID to ACL2.

Note that the soundness of the translation in the other direction, i.e., from ACL2 to UCLID, is *not* trusted, which is the key advantage of their approach. Since UCLID's logic is much simpler than ACL2's, a translator from UCLID to ACL2 will be significantly simpler than a translator in the other direction, and therefore more likely to be sound.

Recently ACL2 has added an external oracle and trust tagging mechanism [21]. This feature could allow for a future version of the UCLID oracle to be built without having to modify ACL2's trusted kernel.

[2] An e-mail inquiry to CVC developers revealed that while their latest SMT solver (named CVC3) has support for proof-generation facilities, the documentation remains sparse [2]. Also of question is the continued support for this feature as CVC itself evolves.

2.6 Alternative to Monomorphisation

As described in Section 4.5, our tool monomorphizes all occurrences of polymorphic types and constants. Couchot and Lescuyer [15] describe an alternative encoding where type variables and type operators are reflected at the term level, and polymorphic constants are then "tagged" with the reflected type instances they occur at. This encoding ends up being significantly more compact than our monomorphization approach, and their benchmarks demonstrate that Yices performs more efficiently on problems translated in this way. One slight drawback is that the translation requires a pair of quantified lemmas to be inserted into the subgoal, which will cause Yices to report even concrete counterexamples as being potentially spurious. However, this could be easily remedied by a separate analysis of the subgoal's decidability that first removes the pair of quantified lemmas. We may consider this translation approach in a future version of `ismt`.

3. HOW THE ISMT TACTIC WORKS

The `ismt` tactic proves theorems by having Yices prove their negations unsatisfiable. When invoked, the `ismt` tactic performs the following tasks on the topmost subgoal of the Isabelle goal stack:

- Translate the types occurring in the formula into Yices type declarations. This process requires monomorphisation of HOL datatypes, records, etc., to map polymorphic operators and type variables to corresponding monomorphic versions supported by Yices. (See Section 4 for details.)
- Negate the subgoal and translate it to Yices. If a HOL constant has no corresponding Yices construct, then declare it as an uninterpreted constant of the appropriate type. For instance, let `isEven :: nat ⇒ bool` be a user defined HOL constant. Then, for the HOL expression `isEven (4::nat)`, we would generate the following Yices translation:

  ```
  (define isEven::(-> nat bool))
  (assert (not (isEven 4)))
  ```

  where no defining equations for `isEven` are added (however the user can always first insert the defining equations into the original subgoal as quantified lemmas).
- Pass the generated script to Yices. If Yices returns a model (i.e., a set of assignments that satisfies the negation of the input), we turn that into a refutation of the original formula. Naturally, the Isabelle proof attempt fails at this point. (This counterexample might be spurious, due to the presence of uninterpreted constants. We will discuss this possibility in detail in Section 5.)
- If Yices determines the clauses are unsatisfiable, then trigger Isabelle's oracle mechanism and accept the original subgoal formula as a (trust-tagged) theorem.

The translator makes no attempt to send existing Isabelle lemmas to Yices. However, the user can always explicitly insert lemmas into the current subgoal as additional hypotheses. (See Section 5.1 for an example.)

The following HOL entities are properly understood and translated by the `ismt` tactic to Yices's internal language. Any other construct will be translated as an uninterpreted constant or type:

- **Types.** Ground types: `int`, `nat`, `bool`. Basic HOL types: polymorphic lists, option type, tuples of arbitrary arity, including `unit`. Records with polymorphic fields (except extensible records). User defined datatype declarations: Both parameterized and recursive variants are supported. (However, they cannot be mutually recursive, either directly or indirectly via nesting.) Functions: Both first-order and higher-order functions are supported.

- **Constants.** Equality: `=`, supported polymorphically at all types. Boolean operators: `True`, `False`, $\leq$, $<$, $\longrightarrow$, $\Longrightarrow$, $\vee$, $\wedge$, $\neg$, and `dvd`. Operators:[3] $+$, $-$, $\times$, $/$, $-$ (unary minus), `div`, `mod`, `abs`, `Suc`, `min`, `max`, `fst`, and `snd`.

- **Expressions and binding constructs:** If-expressions, let bindings, λ-abstractions, quantifiers ($\forall$, $\exists$, $\bigwedge$), case expressions (over tuples, naturals, option type, lists, and arbitrary user defined types), function and record update expressions.

4. EXAMPLES

In this section we will walk over a number of example uses of the `ismt` tactic, demonstrating its basic capabilities.

4.1 Basics

Consider the classic excluded-middle example:

```
lemma "a ∨ ¬a"
by ismt
```

Running Isabelle on this input yields:

```
lemma ?a ∨ ¬?a  [!]
```

Note that Isabelle tags the free variable `a` with `?`, indicating it is a *schematic variable*, i.e., the lemma is proven for all possible substitutions of the variable `a`. Furthermore, the lemma is displayed with a `[!]` annotation, indicating the role of the oracle.

Here's a slightly more interesting example, showing that an odd number can not be a multiple of 2:

```
lemma "a = (2::int) * n + 1 ⟶ a ≠ 2 * m"
by ismt
```

which will be proven by the `ismt` tactic directly. Note that the type qualifier on 2, (i.e., `2::int`) is necessary. Otherwise Isabelle's overloaded numbers would have caused this statement to have a more polymorphic type than intended, and the translator would have left the arithmetic operators uninterpreted.

[3]The arithmetic operators ($+$, $-$, etc.), and comparisons (`<=`, `<`) are supported both at their `int` and `nat` instances. Use of arithmetic operators at other Isabelle numeric types will remain uninterpreted. Also note that Yices does not fully support non-linear arithmetic. If a non-linear expression is given to the translator it will still be translated, but Yices might reject the input.

If the input to the `ismt` tactic is not a Yices-theorem, then a counterexample will be generated. Consider:[4]

```
lemma "abs (n::int) = n"
by (ismt model: abort)
```

Running Isabelle on this input will yield:

```
*** A counter-example is found:
***     n = -1
```

Counterexample generation raises an interesting question in the presence of uninterpreted constants. Consider the following example, where we do not indicate what specific type the expression `None` has.

```
lemma "n = None"
by (ismt model: abort)
```

The tactic will respond with:

```
*** A counter-example is found:
***     Some (ismt_const 1) = n
```

The counterexample uses the function `ismt_const`. The following excerpt from the generated Yices code might help explain the need for this constant:

```
(define-type 'a)
(define-type option-'a
    (datatype None-'a (Some-'a the::'a)))
(define n::option-'a)
(assert (/= n None-'a))
```

The type assigned to the HOL constant `n` is `'a option`, which results in a rendering of the `option` type at the uninterpreted type `'a`. (See Section 4.5 for details on how datatype declarations are translated.) Yices assumes all uninterpreted types are integers when generating models. When the HOL counterexample is generated from the Yices model, however, we cannot use these integers as the values of the corresponding variables; doing so would not be type correct. Hence, we have defined an uninterpreted HOL constant `ismt_const` with the type `int` $\Rightarrow$ `'a`, Such counterexamples should be read such that the arguments to `ismt_const` are indices into the right HOL type, where different indices pick different values in the corresponding domain. (That is, `ismt_const` should be considered an injective function.)

4.2 Anonymous functions

HOL's λ-abstractions are compiled into their Yices counterparts. Here are several examples:[5]

```
lemma "(λx::int. x+2) = (λy. 2+y)"
lemma "(λx::int. x+2) = f"
```

The first lemma generates the following Yices code:

```
(assert (/= (lambda (x::int) (+ x 2))
            (lambda (y::int) (+ 2 y))))
```

[4]The `model: abort` flag instructs `ismt` to throw an exception when a model is returned by Yices. The other possible options are `silent`, which acts as skip; and the default `notify`, which is the same as skip except it also displays the counterexample in Isabelle's trace buffer.

[5]For brevity, we will no longer show the actual call to `ismt`, i.e., each `lemma` line should be followed by the command `by (ismt model: abort)`.

which is automatically proven.

The second lemma generates the following code:

```
(define f::(-> int int))
(assert (/= (lambda (x::int) (+ x 2)) f))
```

which causes the `ismt` tactic to generate the following counterexample:

```
*** A counter-example is found:
***     f -1 = 2
```

Note that the use of λ-expressions can trigger incompleteness, though we have found this to occur rarely in practice. (See Section 6 for details).

4.3 Tuples

HOL tuples are converted to their Yices counterparts appropriately. Tuples of arbitrary arity are implemented by right-nested tuples in HOL, and the translator uses the same technique to represent them in Yices.

The following lemma is automatically proven:

```
lemma "(x, y, x) = (y, x, y) ⟹ x = y"
```

It generates the code:

```
(define-type 'a)
(define x::'a)
(define y::'a)
(assert (= (mk-tuple x (mk-tuple y x))
           (mk-tuple y (mk-tuple x y))))
(assert (/= x y))
```

Projections `fst` and `snd` are translated accordingly:

```
lemma "fst t = snd t ⟹ (snd t, fst t) = t"
```

This lemma generates the following code:

```
(define-type 'a)
(define t::(tuple 'a 'a))
(assert (= (select t 1) (select t 2)))
(assert (/= (mk-tuple (select t 2) (select t 1))
            t))
```

and is automatically proven by Yices.

Any models produced by Yices for the negation of the goal are automatically translated back to HOL. Consider:

```
lemma "snd (f, f True) = False ⟹ f False = True"
```

which generates:

```
(define f::(-> bool bool))
(assert (= (select (mk-tuple f (f true)) 2) false))
(assert (/= (f false) true))
```

We get the following HOL counterexample:

```
*** A counter-example is found:
***     f True = False
***     f False = False
```

Notice that the components of a tuple can contain arbitrary elements, including functions.

4.4 Let expressions

HOL let-expressions are implemented by the higher order function `Let :: 'a ⇒ ('a ⇒ 'b) ⇒ 'b`, so that `let x = 1 in x + x` is syntactic sugar for `Let 1 (λx. x + x)`. We convert these directly to Yices let-expressions. Here is an example:

```
lemma "let x = (1::int) in let y = 2 in x+y = 3"
```

Our tactic generates the following Yices code:

```
(assert (not (let ((x::int 1))
                  (let ((y::int 2))
                       (= (+ x y) 3)))))
```

which is automatically proven.

Bound variables in a let-expression can be of arbitrary types; the translator will ensure appropriate eta-expansion is done to preserve Yices' stringent function arity requirements, as demonstrated below:

```
lemma "(3::int) = (let f = (op +) 2 in f 1)"
```

This lemma will be automatically proven by `ismt`. It generates the following code:

```
(assert (/= 3 (let ((f::(-> int int)
                     (lambda (etav::int)
                             (+ 2 etav))))
                   (f 1))))
```

Note the eta-expansion in the definition of `f` to make sure that the Yices constant `+` is applied to the correct number of arguments.

4.5 Datatype declarations

One of the ubiquitous aspects of functional programming (either in HOL or in any other functional language) is the use of datatype declarations. The `ismt` tactic translates (most) datatype declarations to their Yices counterparts. There are several obstacles, however:

- Yices does not support parameterized or polymorphic datatype declarations. Our translator "flattens-out" the use of parameterized datatypes on the fly, generating individual monomorphised instances.
- Yices does not allow datatype declarations to be mutually recursive, either directly (via the use of two datatype declarations), or indirectly (via the use of nested recursion). Such cases are detected by the translator and rejected.

4.5.1 Datatypes without parameters

Any non-parameterized datatype, recursive or otherwise, generates an equivalent declaration in Yices.

Simple enumerations are converted to `scalar` declarations:

```
datatype Kind = Odd | Even
```

gets translated to:

```
(define-type Kind (scalar Odd Even))
```

Recursive types are translated accordingly:

```
datatype Nat = Zero | Succ Nat
```

gets translated to:

```
(define-type Nat (datatype Zero (Succ aSucc::Nat)))
```

Unlike HOL, Yices requires all fields in a `datatype` declaration to have associated accessors. The accessor `aSucc :: Nat ⇒ Nat` was automatically generated in the last example to satisfy this requirement. (The translator also allows users to register their own custom accessors, see [17] for details.)

4.5.2 Parameterized datatypes

Each use of a parameterized datatype at a different instance causes the translator to generate a new set of declarations. We call this translation the process of *monomorphisation*. To illustrate, consider the following lemma:

```
datatype ('a, 'b) Either = Left 'a | Right 'b
lemma "Left False ≠ Right (4::int)
       ∧ Left True ≠ Right x"
```

causes the translator to generate the following code:

```
(define-type 'a)
(define-type
   Either-bool-int
   (datatype
      (Left-bool-int aLeft-bool-int::bool)
      (Right-bool-int aRight-bool-int::int)))
(define-type
   Either-bool-'a
   (datatype (Left-bool-'a aLeft-bool-'a::bool)
             (Right-bool-'a aRight-bool-'a::'a)))
(define x::'a)
(assert (not (and (/= (Left-bool-int false)
                      (Right-bool-int 4))
                  (/= (Left-bool-'a true)
                      (Right-bool-'a x)))))
```

which is successfully proven by Yices. Note that each distinct use of the `Either` type caused a new datatype declaration, including the case where there is a free type variable. The names of types are used with dashes to create unique constructor names, as in `Left-bool-int` or `Right-bool-'a`.

Recursive declarations are translated similarly:

```
datatype 'a Tree = Leaf 'a
                 | Branch "'a Tree" "'a Tree"
lemma "Leaf 3 ≠ Leaf (2::int)"
```

causes the translator to generate the following:

```
(define-type
   Tree-int
   (datatype (Leaf-int aLeaf-int::int)
             (Branch-int aBranch-int1::Tree-int
                         aBranch-int2::Tree-int)))
(assert (= (Leaf-int 3) (Leaf-int 2)))
```

which is again proven unsatisfiable automatically by Yices.

4.5.3 Direct and nested mutual recursion

Mutually recursive datatypes are not supported by the translator, due to the fact that Yices has no support for such constructs and there is no simple translation that can be applied in such cases. An example of such a declaration is the following recursive pair of datatypes:

```
datatype 'a AExp = Var 'a | BExp "'a AExp"
and      'a BExp = And "'a AExp" "'a AExp"
```

If a lemma involving one of the types `AExp` or `BExp` is sent to `ismt`, it will throw an exception rejecting the mutually recursive datatype declaration.

Nested recursion is another source for the same problem:

```
datatype 'a Term = Variable 'a | App "'a Term list"
```

In this case, the recursion for the `Term` datatype happens at the type `'a Term list` instead of the required type `'a Term`. Similar to the above case, nested recursive declarations will be rejected by `ismt` as well.

4.6 Case expressions

In HOL, every datatype declaration is accompanied by a corresponding case-construct to take the constructed terms apart. Unfortunately, Yices does not support case expressions natively. However, Yices does provide recognizers for each constructor in a datatype declaration, and we take advantage of this facility to compile down HOL case-expressions to a cascaded sequence of if-then-else expressions. Using this technique, we not only support case expressions over built-in types such as tuples, booleans, and lists, but also user-defined datatypes as well. To illustrate, consider the following lemma:

```
lemma "fst tp = (case tp of (x, y) ⇒ x)"
```

which can automatically be proven by `ismt`. It generates the following code:

```
(define-type 'a)
(define-type 'b)
(define tp::(tuple 'a 'b))
(assert (/= (select tp 1)
            (((lambda (x::'a) (lambda (y::'b) x))
              (select tp 1)) (select tp 2))))
```

Note that there is no explicit if-statement used in this case, since there is exactly one way to take a tuple apart.

Case expressions over naturals demonstrates the use of if-expressions:

```
lemma "i+1 = (case (i::nat) of
                 0     ⇒ 1
               | Suc m ⇒ m+2)"
```

Yices can automatically prove the generated assertion for this case:

```
(assert
  (/= (+ i 1)
      (if (= i 0)
          1
          ((lambda (m::nat) (+ m 2)) (- i 1)))))
```

HOL lists are slightly more interesting. Consider:

```
lemma "case [True, False] of
         [] ⇒ True
       | (y#ys) ⇒ y"
```

This lemma is automatically proven, generating the code:

```
(define-type
  list-bool
  (datatype Nil-bool (Cons-bool hd::bool
                                tl::list-bool)))
(assert
```

```
(not
  (let
    ((casev::list-bool
       (Cons-bool true (Cons-bool false
                                  Nil-bool))))
    (if (Nil-bool? casev) true
        (((lambda (y::bool)
                  (lambda (ys::list-bool) y))
           (hd casev))
          (tl casev))))))
```

The translator uses a let-expression to wrap the test expression around (unless it is already a variable), as demonstrated above using the variable `casev`. This aids greatly in readability as it avoids duplicating the expression later on.

4.7 Records

HOL record types are translated into Yices records. HOL's extensible records are not supported, however, since there is no corresponding Yices construct.[6]

Consider the HOL lemma:

```
record pt = pt_x :: int
lemma "pt_x (| pt_x = 3 |) = 3"
```

We generate the following code:

```
(define-type pt (record pt_x::int))
(define-type unit (scalar Unity))
(assert (/= (select (mk-record pt_x::3) pt_x) 3))
```

(Note the appearance of the `unit` type in the output, which seems spurious. It, in fact, corresponds to the `more` field of the HOL record.)

Parameterized and polymorphic fields are converted as usual, by monomorphising them appropriately:

```
record ('a, 'b) pt2 =
  pt2_x :: 'a
  pt2_y :: 'b
  pt2_z :: int

lemma "pt2_x (| pt2_x = v, pt2_y = q,
                pt2_z = s |) = v"
```

The generated Yices code is:

```
(define-type 'a)
(define-type 'b)
(define-type pt2-'a-'b
    (record pt2_x::'a pt2_y::'b pt2_z::int))
(define-type unit (scalar Unity))
(define v::'a)
(define q::'b)
(define s::int)
(assert (/= (select (mk-record pt2_x::v
                               pt2_y::q pt2_z::s)
              pt2_x)
            v))
```

HOL and Yices records are both extensional, allowing us to prove record equality theorems. Consider the following lemma that uses the `pt2` record as defined above:

[6]We plan to overcome this limitation in a future version by "flattening," i.e., either by compiling HOL's extensible records to Yices records that contain all the relevant fields, or by compiling them into nested records.

```
lemma "(| pt2_x = a, pt2_y = b, pt2_z = a+b |)
       = (| pt2_x = b, pt2_y = a, pt2_z = c |)
  ⟹ a = b & c - b = a"
```

It generates the following additional code:

```
(define a::int)
(define b::int)
(define c::int)
(assert
  (= (mk-record pt2_x::a pt2_y::b pt2_z::(+ a b))
     (mk-record pt2_x::b pt2_y::a pt2_z::c)))
(assert (not (and (= a b) (= (- c b) a))))
```

which is proven automatically by Yices.

Finally, counterexamples will be translated back to their HOL counterparts:

```
record fr =
  f :: "int => int"

lemma "r1 = (r2 :: fr)"
```

A call to `ismt` generates the following counterexample:

```
*** A counter-example is found:
***     (| f = f r1 |) = r1
***     (| f = f r2 |) = r2
***     f r1 1 = 2
***     f r2 1 = 3
```

4.8 Function and record updates

HOL's function and record update notations are fully supported by the translator. Consider the following trivial function update theorem:

```
lemma "(f(i:=n)) i = n"
```

This lemma is successfully proven. It generates the following code:

```
(define-type 'a)
(define-type 'b)
(define f::(-> 'b 'a))
(define i::'b)
(define n::'a)
(assert (/= ((update f (i) n) i) n))
```

Record updates are similarly translated to their Yices equivalents.

4.9 Quantifiers

Quantifiers are a soft spot for SMT solvers, as they typically render the underlying algorithms incomplete. (We will return back to this point in Section 6 in detail.)

Our `ismt` tactic translates both meta- and object-level Isabelle quantifiers into Yices' input format. One optimization is that top-level universally bound variables are skolemized into top-level Yices uninterpreted constants. To illustrate, consider the following trivial lemma:

```
lemma "⋀ x. (∀y. (x = x ∧ y = y))"
```

When `ismt` is invoked, it generates the following code that contains no quantifiers at all:

```
(define x::'a)
(define y::'b)
(assert (not (and (= x x) (= y y))))
```

Needless to say, Yices deduces unsatisfiability instantly.

When quantifiers are nested, however, the translator can no longer compile them away. In such cases, we simply translate them to their Yices equivalents. Consider the lemma:

```
lemma "⋀x. ⟦ (∀y. p y ⟹ q y) ⟧ ⟹ p x ⟶ q x"
```

This lemma is proven successfully by Yices. The generated code is:

```
(define-type 'a)
(define x::'a)
(define p::(-> 'a bool))
(define q::(-> 'a bool))
(assert (forall (y::'a) (=> (p y) (q y))))
(assert (not (=> (p x) (q x))))
```

Note, in particular, how the parameter `x` becomes a top-level definition, while `y` remains `forall` bound in the Yices translation.

The treatment of the ∃ binder is similar, except that top-level occurrences cannot be compiled away to top-level definitions. Here is a simple example to illustrate:

```
lemma "∃x. x > (0::nat)"
```

This lemma is proven automatically by Yices. It generates the code:

```
(assert (not (exists (x::nat) (< 0 x))))
```

The translator does not support the unique-existence quantifier (∃!). The bounded versions of the quantifiers (`Ball` and `Bex`) are not supported either, and neither are the Hilbert's choice (ϵ), and the `Least` binders. Uses of these constructs will remain uninterpreted during the translation process. (It might be possible to support bounded quantifiers through Yices' predicate-subtyping. However, we currently refrain from this since Yices does not ensure type-correctness when predicate-subtyping is used. In particular, it is possible to define empty types in Yices, and exploit these to prove bogus theorems.)

5. DEALING WITH FALSE ALARMS

Due to the fact that certain constants will remain uninterpreted during the translation, the `ismt` tactic can come up with bogus counterexamples. In this section we consider two particular instances of this problem and discuss mitigations.

5.1 Recursive uninterpreted constants

Although non-recursive uninterpreted functions can be dealt with by unfolding their definitions before calling `ismt`, a different approach is needed when the functions are recursively defined. To illustrate, consider the function `len` below, which computes the length of boolean lists:

```
consts len :: "bool list ⇒ nat"
primrec
   "len []     = 0"
   "len (x#xs) = 1 + len xs"
```

Consider the following lemma:

```
lemma "len [True, False] = 2"
```

which generates the following code:

```
(define-type list-bool
    (datatype Nil-bool
              (Cons-bool hd::bool tl::list-bool)))
(define len::(-> list-bool nat))
(assert (/= (len (Cons-bool true
                    (Cons-bool false Nil-bool)))
            2))
```

Yices provides the following counterexample:

```
*** A counter-example is found:
***     len [True, False] = 3
```

which is clearly bogus. The problem arises since we have not told Yices anything about the function `len`, leaving it uninterpreted.

There are clearly easier ways to prove this lemma in Isabelle, (in fact, a simply application of `auto` would suffice), but our goal is to show how additional quantified hypotheses can be added so that `ismt` can prove the lemma successfully. In this case, all we need to do is to assert the pattern-matching rewrite rules for `len` as extra Isabelle lemmas:

```
lemma len0: "len [] = 0"
lemma len1: "len (x#xs) = 1 + len xs"
```

Both of these lemmas can be proven by Isabelle's `auto` tactic. We can now use these additional facts to guide Yices:

```
lemma "len [True, False] = 2"
apply (insert len0 len1)
```

The goal state after the `insert` tactic looks like:

```
⟦ len [] = 0; ⋀x xs. len (x # xs) = 1 + len xs ⟧
 ⟹ len [True, False] = 2
```

When we apply `ismt` at this proof state, the generated Yices code looks like:

```
(define-type
  list-bool
  (datatype Nil-bool (Cons-bool hd::bool
                                tl::list-bool)))
(define len::(-> list-bool nat))
(assert (= (len Nil-bool) 0))
(assert
  (forall (x::bool)
    (forall (xs::list-bool)
            (= (len (Cons-bool x xs))
               (+ 1 (len xs))))))
(assert (/= (len (Cons-bool true
                    (Cons-bool false Nil-bool)))
      2))
```

which is easily decided by Yices to be unsatisfiable, allowing us to conclude that the original formula is indeed a theorem.

Unfortunately, not all false alarms can be dealt with using these techniques. There will invariably be certain constructs that will go uninterpreted during the translation. (Consider, for instance, more complicated recursive definitions where finding such "helper lemmata" would amount to proving the original theorem. Or HOL constructs such as Hillbert's choice operator that has no corresponding "executable" counterpart that we can use in the simplification process.) While these techniques can be helpful, our experiences with the `ismt` tactic suggest that such cases are best dealt within the theorem proving framework of Isabelle, instead of relying on a backend SMT solver.

6. INCOMPLETENESS

SMT solvers typically support richer languages/logics than they can actually decide. For instance, it is well known that quantifiers (i.e., $\forall$, $\exists$), and λ-expressions make logics incomplete. In such cases, the underlying solver typically returns a satisfying model as well, but there is a chance that this model might be bogus. (Such problems are reported by Yices as "unknown" to indicate this possibility.)

To illustrate, consider:

```
lemma "((λ(x::bool). f x) = (λx. True)) ⟹ f x"
```

The generated file contains (excerpt shown below):

```
(assert (= (lambda (x::bool) (f x))
           (lambda (x::bool) true)))
(assert (not (f x)))
```

We get the following "potential" counterexample from `ismt`, which is actually bogus in this particular case:

```
Potential HOL counterexample:
    x = False
    f False = False
```

Notice that incompleteness will *never* cause `ismt` to prove a non-theorem. Rather, it might prevent it from proving a valid assertion. In other words, soundness is never at risk due to this limitation.

7. EXPERIMENTS

Having discussed the `ismt` tactic in detail, we will now briefly turn to our use cases for it at Galois, providing examples from various projects, both past and present.

Galois is building several *cross-domain* web service applications that provide strong security guarantees about the confidentiality of data across separate security domains. For example, our *Trusted Services Engine* (TSE) is a multi-level secure filestore providing a single common repository for files and directories, where each user's view of the file system is restricted according to that user's security level.

Most of the TSE is written in the Haskell functional programming language, which provides a number of security benefits. However, we implemented the most security-critical cross-domain portion of the TSE as an 800-line C program, to eliminate any dependency on Haskell's runtime system. In order to attain the highest government security certifications for the TSE, we decided to formally model this component in Isabelle and verify its memory safety and information flow properties [19]. Although the verification was successful, discovering the required inductive safety invariants and view relations was very labor-intensive.

7.1 C program verification

To speed up proofs for future cross-domain C components we have prototyped a monadic-style sequential semantics for a larger subset of C, called `SeqC`. Although `SeqC`'s semantics still only covers a small portion of the C standard, it includes general `while` loops, the non-local control flow statements `return`, `break`, and `continue`, access control permissions for each memory byte, the ability to take addresses of C globals, locals, and malloc'ed memory, nondeterministically-modeled C functions such as `malloc` and `free`, and general `assume` and `assert` statements. `SeqC` also contains a non-recoverable `Err` state that is transitioned into upon any memory-safety violation or assertion failure. Memory-safety is thus defined as unreachability of `Err`, with `assume` statements first pruning out all execution paths that don't satisfy a program's environmental assumptions.

We have proved Hoare logic rules for `SeqC`, built a simple verification condition generator (VCG) as an Isabelle tactic, and run some initial experiments verifying memory-safety of small example C programs. So far we have found the `ismt` tactic to be quite helpful not only in proving the verification conditions, but also for debugging too-weak preconditions and loop invariants by inspecting the counterexamples returned. To keep the counterexample sizes tractable, we initially put small concrete bounds on program parameters such as array sizes. We then used selected rewrite rules and a high-level Isabelle tactic to automatically expand away the (now bounded) quantifiers and recursive functions in the formula before calling `ismt`. Once we had found the appropriate rewrites, preconditions, and loop invariants, the verification of our example programs was completely automatic, thanks to the versatility of our `ismt` tactic and the power of Yices.

To discover the necessary rewrite rules, we first had to figure out which part of the verification condition `ismt` couldn't solve. We wrote an Isabelle proof script that used case-splitting tactics to eliminate any rigid quantifiers (i.e. quantifiers that didn't require any witnesses to be invented), as well as top-level conjunctions and disjunctions in the proof goal. The result was one or more smaller subgoals that were jointly equivalent to the original proof goal. Then, we used the `ismt` tactic on each subgoal until it found one that failed.

7.2 Parameterized program verification

We have also used the `ismt` tactic in verifying a string-copy routine where the source string can either be on the heap or the C stack, with a precondition that the string length is not larger than the destination string buffer, nor aliased to the destination buffer or any of the program's local variables.

When we tried to re-run the verification where the source and destination buffer sizes were not fixed beforehand, the resulting goals required Yices to reason about quantified assertions. Unfortunately, we have found that Yices' quantifier instantiation heuristics were not up to the task. Furthermore, Yices currently does not allow users to specify their own domain-specific quantifier instantiation term patterns.

To illustrate the issues we encountered, below we have defined `vcg`, a simplified version of one of the parameterized subgoals generated by our case-splitting tactic that Yices was not able to solve, even with the appropriate quantified lemmas.[7]

```
definition
   vcg :: "addr ⇒ addr ⇒ addr ⇒ int ⇒
           (addr ⇒ byte) ⇒ bool" where
  "vcg src dst s_ptr n h
   = (let s = h s_ptr;
          d = dst - src + s;
          h' = h(d := h s, s_ptr := h s_ptr + 1)
      in ( src ≤ s ∧ is_str s (src + n - s) h
         ∧ ¬ s_ptr mem (str_addrs s n h)
```

[7]The combined Isabelle and `ismt` proof of this formula is part of a self-contained example file `vcg.thy`, included in our `ismt` release [5].

```
∧ ¬ d mem (str_addrs s n h)
∧ h s ≠ 0
⟶ ¬ s_ptr mem (str_addrs (s+1) n h'))"
```

Parameter `src` points to the start of the source string buffer, `dst` points to a destination buffer of size `n`, and `s_ptr` is `&s`, the address of the C variable `s`, which itself points to the next byte to copy from the source string. The variable `h` is the contents of the memory heap at the top of the loop. In the formula we have defined `d` to point to the destination byte that `*s` will be copied to, and `h'` to be the updated heap at the bottom of the current loop iteration where `*s` has been copied to `*d` with `s` incremented.

The function `vcg` mentions three recursively-defined functions. (i) The predicate `is_str` has type `addr ⇒ int ⇒ (addr ⇒ byte) ⇒ bool`; `is_str p n h` is true whenever `p` points to a null-terminated string in heap `h` that is no more than `n` bytes long, including the null byte. (ii) The function `str_addrs` has type `addr ⇒ int ⇒ (addr ⇒ byte) ⇒ addr list`. The call `str_addrs a n h` returns a max-length `n` list of contiguous addresses in `h` starting at `a` up to and including the first address pointing to a null byte (if any). It represents the set of addresses that can alias the string. And (iii), the infix binary relation `_ mem _ :: 'a ⇒ 'a list ⇒ bool` is list membership. The types `addr` and `byte` are synonyms for `int`.

To make this example easier for `ismt`, we manually removed from `vcg` any hypotheses that were irrelevant to the conclusion. The remaining hypotheses are: At the top of the while loop `s` points to a string that fits within the `n`-byte source buffer `src`, `&s` and `d` are not aliased to the string, and `*s` is not null. The conclusion to verify is that at the end of the while loop `&s` is still not aliased to the string now pointed to by `s`. (This could happen if the while loop overwrites the source string's null byte.)

Discovering quantifier instantiations. We needed to give `ismt` four quantified lemmas for it to verify the formula. However we had to manually instantiate the following lemma with the substitutions {`p' ← d, x ← h s`} and {`p' ← s_ptr, x ← s + 1`}, to keep `ismt` from timing out. The lemma variables `p`, `n`, and `h` remained quantified. With these instantiations, `ismt` is able to verify `vcg` in less than a second.

```
lemma str_addrs_simp:
 "¬(p' mem str_addrs p n h) ∨ ((h p'=0) = (x=0))
  ⟹ str_addrs p n (h(p' := x)) = str_addrs p n h"
```

To find the required lemmas and instantiations, we started a manual backchaining process, where we asserted additional formulas as new Isabelle subgoals that we believed to be true. We confirmed this by running ismt on `vcg`, with the additional formulas as extra hypotheses. We then recursively followed this debugging process on each new Isabelle subgoal, until it was clear what extra lemma or instantiation was needed. (The file `vcg.thy` included in the distribution contains the additional formulas we used, specifically in the proof of `detailed_vcg_lemma` [5].)

The counterexamples generated by `ismt` were helpful for debugging these subgoals if they were small enough. However, we often found it quicker to ignore the counterexample and instead inspect the abstract Isabelle subgoal. This was because Yices would typically assign multiple variables the same concrete value in the counterexample, e.g., `x = 3 ∧ y = 3`. If the counterexample also asserted a formula (P `3`) that we knew to be false, we couldn't then tell whether Yices had chosen to satisfy (P `x`) or (P `y`). But this information was usually necessary to determine which quantified lemma would eliminate the counterexample for all possible values of `x` or `y` in the subgoal. It would be very helpful to have either a Simplify-style label capability that returned which subpropositions actually contributed to the counterexample, or else a command to return an "abstract" counterexample that displays the occurrences of free variables in formulas and subterms, rather than substituting in their concrete values.

8. TIPS FOR USING ISMT

The following list summarizes a number of tips reflecting our experiences with the `ismt` tactic.

- *Avoid nested quantifiers.* The translator will generate separate top-level `assert` statements for each quantified hypothesis found in the subgoal. This is preferable as it seems to enable more of Yices quantifier instantiation heuristics. Try to lift quantified formulas into top-level hypotheses whenever possible.

- *Restrict arithmetic to* `nat`*'s and* `int`*'s.* While Isabelle allows arithmetic over arbitrary types (using axiomatic type-classes), the Yices backend is not rich enough to understand such constructs. When such goals are sent to `ismt`, it is very likely that a bogus counterexample will be returned, since these number types will remain uninterpreted. Try to restrict arithmetic to `nat`'s and `int`'s only, which are fully supported.

- *Watch for "uninterpreted" constants.* Pay close attention to the counterexamples returned by `ismt`.[8] It is likely that the falsifying model will be bogus due to unknown facts about these constants. If possible, apply the techniques described in Section 5 to resolve such cases.

- *Be type-specific, especially when using records.* Isabelle rewrite rules regarding records tend to be too polymorphic, applying to a variety of record types. Since the translator does not support extensible records (see Section 4.7), such rewrite rules can create subgoals that are unnecessarily more general than needed. Whenever possible, provide sufficient type annotation in the HOL specification to restrict the record types appropriately.

- *Dealing with "potential" counterexamples.* Heavy use of quantifiers tend to render the underlying SMT solvers incomplete. Isabelle's `safe` tactic might help in cases where such quantifiers can be eliminated.

9. CONCLUSIONS

In this paper, we have described our Isabelle `ismt` tactic. The tactic provides an automated solver for a subset of HOL, by translating them appropriately to Yices. As a proof-of-concept work, the `ismt` tactic demonstrates that it is quite

[8]To aid in this process, `ismt` will always print the list of uninterpreted constants that are used in the counterexample.

feasible to build highly useful interfaces to modern off-the-shelf SMT solvers from within interactive theorem proving assistants.

Not every HOL theorem can be proved by `ismt`. Logics of modern SMT solvers are deliberately weaker, trading expressive power for decidability. Still, we have found that the use of the "uninterpretation" technique to translate literally all HOL theorems to such logics pays-off very nicely in practice. Many tedious theorems can be proven "with the push of a button."

We also have several items on our Yices wish-list, as well as enhancements we would like to make to `ismt`, as we discuss below.

Quantifier instantiation term patterns. As described in Section 7.2, Yices' in-built quantifier instantiation heuristics did not always work for our use cases. The Simplify and Z3 SMT solvers support quantifier instantiation term patterns, where the user can attach a set of *instantiation patterns* to any quantified assertion [12]. Each instantiation pattern contains a set of term patterns, and a substitution over the term patterns' free variables. If in its proof search the SMT solver finds a set of subterms that jointly matches the term patterns, it will instantiate the quantified assertion according to the substitution and add it to the proof context.

Instantiation patterns can be used to enable systems of rewrite rules by inserting each rewrite as a quantified equality and an instantiation pattern that matches the left-hand-side of each rule. Instantiation patterns can also implement forward- or backward-chaining inference rules, by having the term pattern match either the hypotheses or conclusion of a quantified implication.

For instance, in Section 7.2, we could have given a single instantiation pattern for lemma `str_addrs_simp` that would have solved the subgoal as well as related subgoals:

```
term_patterns: "¬(p' mem str_addrs p h h)",
               "str_adders p n (h(p' := x))"
substitution: p ← p, p' ← p', n ← n, x ← x.
```

which would have simplified the proof significantly.

Abstract counterexamples and call-back tactics. The ability to generate counterexample information is helpful, especially when the counterexamples are small enough (i.e. have only a few free variables and uninterpreted constants). However, we think it would be much more useful to have a Yices command to return an *abstract* counterexample formula, as described in Section 7.2.

Abstract counterexamples would be especially useful in conjunction with Yices' incremental API. In particular, in future work we want to allow the user to attach Isabelle *call-back tactics* that are invoked on the abstract counterexamples produced by Yices. The call-back tactics would then analyze each counterexample and return lemmas that either refute it or infer additional facts such as equalities. These lemmas would then be incrementally asserted into Yices' current proof context the proof search would resume.

This process would continue until Yices is either able to prove the original formula or else produce a counterexample that the call-back tactics could not infer anything about. Call-back tactics would thus allow the user to safely write domain-specific quantifier instantiation heuristics that could be more powerful than a fixed collection of term patterns.

A further Yices optimization would be to export the E-graph matching API. Users could then register their own term patterns. When a pattern matches during the proof for the first time, Yices would immediately suspend and return references to the matched subterms. There would also have to be commands to query the current proof context (partial model). Exporting the E-graph matcher would also allow call-back tactics to be triggered early, without having to wait for a full counterexample to be built.

Modes of integration. The only mode of integration we have investigated in this work is the oracle mode; where the backend solver and the bridge code is blindly trusted. In order to stay within the pure-LCF style, a proof-generating backend solver, and a proof-replaying (or proof-checking) interface is needed.

Integration with other SMT solvers. The `ismt` tactic has been designed such that other solvers can be plugged in by providing appropriate translators. Currently, we only have a backend for Yices, however. Having translators for multiple backends would make our tactic more useful in the long run, as one can pick the best solver for the task at hand as appropriate. It would of course simplify matters greatly if there was a common SMT language that we could target once and for all. Although SMT-Lib would seem to be a natural target, it currently does not satisfy our needs. In particular having support for recursive datatypes is essential. Also support for an incremental API and a counterexample format would be very helpful. Grundy et al. provides a nice desiderata for such a specification language [20].

Support for further HOL constants. Currently only a subset of HOL constants are "interpreted," (those we found useful in our own experience). While this subset is fairly large, supporting more constants would be useful. In particular, devising a more general scheme to translate all non-recursive HOL definitions to Yices equivalents would be desirable. Paired with a simple mechanism to allow users to indicate which constants should be translated, the `ismt` tactic can act as a powerful tool in custom theory development.

Support for further HOL constructs. The translator currently does not support extensible records. Also, rationals and other numeric types (i.e. those other than `int` and `nat`) go uninterpreted during the translation as well. Adding support for these constructs might prove useful in certain application areas.

Parallel proof processing. Our `ismt` implementation does not support Isabelle's recent multi-core code architecture changes that support parallel proof processing. Thus `ismt` must be called in a single-threaded context. Supporting parallel proofs where each thread might call `ismt` independently would be a nice addition.

Acknowledgments

We would like to thank SRI's Bruno Dutertre and Natarajan Shankar for answering numerous questions on Yices, Alwen Tiu of Australian National University for discussions on the `smt` tactic, and Lee Pike and anonymous reviewers for comments on an earlier draft of our paper.

Availability

Our `ismt` tactic, along with a user's guide and other supporting material is freely available on the internet with a permissive BSD-style license [5].

10. REFERENCES

[1] CVC Lite web site. `www.cs.nyu.edu/acsys/cvcl/`.

[2] CVC3 web site. `www.cs.nyu.edu/acsys/cvc3`.

[3] haRVey web site. `harvey.loria.fr/`.

[4] HOL Lite web site. `www.cl.cam.ac.uk/~jrh13/hol-light/`.

[5] The `ismt` tactic web site. `www.galois.com/company/open_source/ismt`.

[6] PVS: Specification and Verification System site. `pvs.csl.sri.com/`.

[7] Satisfiability Modulo Theories Solvers. `combination.cs.uiowa.edu/smtlib/solvers.html`.

[8] SMT-LIB logics web site. `combination.cs.uiowa.edu/smtlib/logics.html`.

[9] SMT-LIB web site. `combination.cs.uiowa.edu/smtlib/`.

[10] SMT oracle web site. `users.rsise.anu.edu.au/~tiu/smt/`.

[11] Yices web site. `yices.csl.sri.com/`.

[12] Z3 web site. `http://research.microsoft.com/projects/z3/index.html`.

[13] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *PDPAR'03 Workshop, Miami, Florida*, July 2003.

[14] D. Barsotti, L. P. Nieto, and A. F. Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Electr. Notes Theor. Comput. Sci.*, 145:63–78, 2006.

[15] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21st International Conference on Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Computer Science*, pages 263–278, 2007.

[16] B. Duterte and L. de Moura. The YICES SMT Solver. Available at: `yices.csl.sri.com/tool-paper.pdf`.

[17] L. Erkök. Connecting SMT solvers and Isabelle: The `ismt` tactic and the yices bridge. Technical Report, Available upon request from Galois.

[18] P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[19] P. Graunke. Verified safety and information flow of a block device. In *Systems Software Verification (SSV 08) Workshop*, Sydney, Australia, Feb. 2008.

[20] J. Grundy, T. F. Melham, S. Krstić, and S. McLaughlin. Tool building requirements for an API to first-order solvers. *Electr. Notes Theor. Comput. Sci.*, 144(2):15–26, 2006.

[21] M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber. Integrating External Deduction Tools with ACL2. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on Implementation of Logics (IWIL 2006)*, volume 212 of *CEUR Workshop Proceedings*, pages 7–26, Phnom Penh, Cambodia, Nov. 2006.

[22] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[23] P. Manolios and S. K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 855–862, Washington, DC, USA, 2005. IEEE Computer Society.

[24] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR'05 Workshop*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, Jan. 2006.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

Automated Proof with Caduceus: Recent Industrial Experience

Daniel Sheridan
Adelard LLP
College Building
10 Northampton Square
London, UK
djs@adelard.com

ABSTRACT

Small embedded computing devices such as smart sensors replace simple and straightforward analogue devices in industrial applications, extending their functionality to provide additional configurability and features. They are simpler to understand and deploy than programmable controllers, but nevertheless bring a high level of hidden complexity in the form of their embedded software, microcontroller and internal storage. As part of an ongoing series of projects looking at gaining confidence in the correctness of these devices, we have attempted correctness proofs of parts of a real device. We have had considerable success applying the verification tool *Caduceus* to this industrial code with minimal modifications. In this paper, we summarise the approach we have taken, our results, and our opinions about the current state of the available tools.

Categories and Subject Descriptors

F.3.1 [**Information Systems Applications**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*

General Terms

Experimentation

Keywords

Automatic verification, Caduceus, smart sensors

1. INTRODUCTION

"Smart" devices such as smart sensors are specialised embedded computer devices which reproduce the functionality of traditional analog industrial transmitters and alarms. A smart sensor could interface a thermocouple temperature probe to a standard 4–20mA transmission line, while providing scaling and linearisation of the signal, and rate-of-change and level alarms on separate outputs. These specialised but flexible devices are the subject of considerable interest in the nuclear industry at the moment, due to their ability to directly replace simple and straightforward analogue devices, while extending their functionality to provide additional configurability and features.

Smart sensors present users with the usual problems of deploying computer-based devices in safety-critical applications: testing provides insufficient confidence in the correct behaviour of the device, but manufacturers are unwilling to commit to formal development methods to achieve demonstrable correctness. We are involved in the alternative approach of trying to provide maximum post-hoc confidence in the device's behaviour at minimum cost.

Adelard has been fortunate in obtaining the source code of a widely-used smart sensor, which we have been using as a case study for a number of research projects for the nuclear industry. Smart sensors turn out to be particularly amenable to several verification techniques due to the relative simplicity of their core functionality.

In this paper, we describe our experiences with the automated verification tool *Caduceus* [7] in mechanising the verification of part of this code. This is code that, in the past, has turned out to be challenging for analysis tools, as it has not been written with verification in mind. The code is written for compactness, not ease of analysis; on the other hand it is functionally fairly straightforward, making it amenable to formal specification.

1.1 The Caduceus/Why tools

Why [7] is a verification condition generator which operates on a specialised functional language. It is capable of generating verification conditions for a wide range of proof assistants (including, Isabelle, PVS and Coq) and decision procedures (including Yices, Z3 and CVC3), and provides a number of useful features including a graphical interface for executing and collating results from several decision procedures simultaneously.

Why is packaged with *Caduceus*, a front-end for verifying C programs, and *Krakatoa*, a front-end for verifying Java programs. Both tools adopt a JML-like syntax for adding preconditions, postconditions and invariants as inline comments in the code (see [4] for more details on JML).

Caduceus is particularly significant, as there are few formal verification tools which can accept significant subsets of C. We discuss how we worked around the remaining limitations of Caduceus' C parser in Section 2.2.

2. THE SMART SENSOR EXAMPLE

The commercial off-the-shelf smart sensor which has been the subject of this case study is the latest in a line of similar devices from the same manufacturer[1]. The device is capable of reading a process variable from a variety of sources including temperature sensors, potentiometers, and current sources. Profiles for several types of sensor are built in to the preprocessing code. The device has limit and rate-of-change alarm outputs, as well as a scaled and linearised analog output, and it is configurable through an interactive menu and via a serial port.

In an earlier research project, we carried out a manual formal proof of the "core" of the smart sensor software, following [9]. The intention of restricting the proof to the core of the software was both to reduce costs and to focus effort on the most critical part of the code. In addition to the usual risk of incorrect specifications, there are two risks with this approach: unidentified faults in the unexamined code, and mistakes in the proof itself. A later project used a series of non-interference arguments to justify not requiring a detailed proof of the remaining code. The menu-driven configuration interface in particular was subject to a novel correctness argument [3] based on the claim that the menu only set configuration variables to the values chosen by the user.

The core code consists of four functions and roughly 150 non-comment lines of code.

Manual proof turns out to be quite efficient for the type of code in the case study, but it is unlikely to scale well to larger systems, especially as one of the main costs was in presenting the proofs for the client. It is important not to overlook the potential for mistakes to be made in the proof, and how hard it is to review this type of work. In contrast, applying theorem provers can be quite time consuming, but has the advantage of repeatability and the relative assurance of correctness.

The Caduceus/Why tool set has another attraction for the nuclear industry in particular: diversity of solvers. Diversity (separate implementations of the same functionality) is a well-accepted technique for producing high-reliability systems, so the prospect of diverse backends to Why improving the confidence in the verification outcome was met with enthusiasm by our clients.

2.1 Developing specifications

Although we had access to the software design documents for the case study, constructing appropriate specifications was something of a challenge: no formal specification was given in the documentation. We were able to construct a plausible set of specifications from the user documentation, design documents, code comments and, in some cases, the code itself. Developing the specification took a sizable fraction of the manual proof time.

The core code makes the following transformations to the process variable T:

Trimming: Adjust the internal representation of the process variable to match the physical value—intended as a calibration function. There are four expected outcomes:

1. If single-point trimming is enabled, $T' = T + T'_l - T_l$
2. If two-point trimming is enabled, $T' = (T - T_l)\frac{T'_u - T'_l}{T_u - T_l} + T'_l$
3. If two-point trimming is enabled, but the unit is misconfigured ($T_u = T_l$), report a divide-by-zero error, and set $T' = T$
4. If trimming is not enabled, $T' = T$

where variables T_u, T_l, T'_u, T'_l are user-configurable constants indicating the upper and lower input and output values.

Scaling: Ostensibly the same function as trimming, but with a different presentation to the user: it is intended to convert the internal representation to the user's chosen units. Although the specification is similar, the code is quite different from the trimming function. Scaling has the additional function of clamping the output to a pre-defined valid range. There are three expected outcomes:

1. If scaling is enabled, $T' = (T - T_f)\frac{T'_f - T'_z}{T_f - T_z} + T'_f$ and $T_{\min} \le T' \le T_{\max}$, or $T' = T_{\min}$, or $T' = T_{\max}$
2. If scaling is enabled, but the unit is misconfigured ($T_f = T_z$), report a divide-by-zero error, and set $T' = \max(\min(T, T'_{\max}), T'_{\min})$
3. If scaling is not enabled, $T' = T$

where variables T_f, T_z, T'_f, T'_z are user-configurable constants indicating the full-scale and zero-scale input and output values, and $T'_{\max}$ and $T'_{\min}$ are the maximum and minimum permitted results.

Linearisation: As a more flexible alternative to scaling, the sensor can be programmed with a "linearisation" curve allowing arbitrary functions to be applied to the process variable. Linearisation arrays L (giving the input values) and L' (giving the output values) are defined by the user. The code searches for n such that either

- $L[n-1] \le T < L[n]$ (interpolation is applicable)
- $L[n] < T \wedge n = |L|$ (upper range extrapolation is applicable)
- $T < L[n-1] \wedge n = 1$ (lower range extrapolation is applicable)

[1]Unfortunately, commercial sensitivities prevent us from being able to identify the device or the manufacturer.

and applies the transformation
$T' = (T - L[n-1]) \frac{L'[n]-L'[n-1]}{L[n]-L[n-1]} + L'[n-1]$,
raising a divide-by-zero error if $L[n] = L[n-1]$.

The search part of the linearisation function is implemented as a binary search, which narrows the range of indices to 10, followed by a linear search to find the exact position, so we were required to generate invariants for these loops in addition to the specification (we did not attempt to prove termination in this case).

2.2 Adapting source code

Caduceus acts directly on C source code, but necessarily has some restrictions on the code that it accepts: it does not support strings, unions or gotos. We therefore had to make small modifications to the case study source code so that Caduceus could handle it.

Any changes to the code which is to be verified raise the question of traceability: are the results of verifying the modified code still applicable to the original code? We were able to (informally) argue that each of our changes were small enough to not be a problem, but this may be more difficult on larger and more complex code bases.

The code changes required were quite straightforward, and are described below.

2.2.1 *References to unnecessary header files*

The code includes references to stdio.h and math.h:

```
#include <stdio.h>
#include <math.h>
```

These header files seem to have been included out of habit—there is no reference to any of the defined functions or macros in the code, and the code continues to compile without the includes. As the headers use types that cannot be handled by Caduceus, we removed both includes.

2.2.2 *Version control string*

A common idiom when keeping source code in a version control system is to include a string constant in the source which is automatically populated by the version control system, providing traceability even for the binary.

```
const char *VersionString = "VER    1.1 ";
```

This string constant is never read by the code, so can be removed.

2.2.3 *Replace direct memory accesses with arrays*

Arbitrary memory accesses are inevitably prolematic. The sensor software uses an in-memory table (actually, in-memory-mapped-EEPROM) for the custom linearisation function:

```
float ReadPntVal(char PntNo, int Base)
{
    float *Address;
    Address = (float *)(Base + ((int)PntNo * 4));
    return *Address;
}
```

We can model this as an array lookup:

```
float LinearisationTable[2048];
```

[...]

```
/*@ ensures
    \result == LinearisationTable[PntNo + Base]
 */
float ReadPntVal(char PntNo, int Base)
{
    return LinearisationTable[PntNo+Base];
}
```

Conveniently, the base addresses used in the code are close to the beginning of the memory space, so we can simply model an 8Kb sequence of floats positioned at the start of memory. If the array `LinearisationTable` were actually stored at the start of memory, the two definitions of ReadPntVal above would be equivalent, due to the way C is defined (see 6.5.2.1 of IEC/ISO 9899 [8]).

2.2.4 *Remove use of unions from Linearise Value*

This union structure is used as the data type for linearisation table entries:

```
union FloatToByte
{
    BYTE    B[4];
    float   F;
    long    L;
};
```

This allows for data type conversions to be performed away from the watchful eye of the C type checker. It is only used once:

```
if ((LinearPtEX.L == 0xffffffffL) ||
    (LinearPtEY.L == 0xffffffffL))
{
    ErrorStatus |= ERROR_BAD_CUSTOM_CURVE;
    break;
}
```

If we remove this block, all remaining accesses to variables of type FloatToByte are to the float member only; we can therefore replace this data type with an ordinary float.

This is the only behaviour-changing modification that we made, since we have removed the ability to detect certain types of corrupted table. In fact, this condition seems to be intended to detect an uninitialized EEPROM; if we take appropriate configuration as a precondition for the analysis, the code can be safely ignored.

2.2.5 *Remove all unions*

The union types defined in the other header file are not used in the analysed code (after making the replacement described above). It is therefore safe to remove all union type definitions.

2.3 Developing annotations

Having already conducted a manual proof of this code, we had already developed pre- and postconditions for each function. These needed to be translated into the annotation language for Caduceus. A development process was adopted for constructing the annotations:

- A proof was attempted with no pre/post conditions. This helped identify code that Caduceus could not parse, and highlighted the divide-by-zero issue discussed below.
- A proof was attempted separately for each precondition/postcondition pair. This meant that mistakes and problems were discovered sooner, and were easier to trace to their sources.
- A combined postcondition was created containing all of the cases, so that verification could be carried out in a single step.

2.4 Experience of applying Caduceus

We tackled each of the three functions separately, and in each case, verification was attempted using the SMT solver CVC3 [2], generating input for it using the CVC-lite and SMT-LIB backends of Why, and with the SMT solvers Yices [6] and Ergo [5]. Example solver output for Trimming are shown in Table 1[2].

In some cases, we were able to prove more things using the more expressive CVC-lite encoding than with the SMT-LIB encoding, while in other cases the SMT-LIB encoding was more successful. The main difference that we observed was in the encoding of division: the SMT-LIB encoding uses an uninterpreted function to represent division, preventing the decision procedure from manipulating the arithmetic. In some cases, where the division itself was not required to reach the conclusion of the proof, the use of uninterpreted functions allowed CVC3 to obtain a proof sooner.

Yices was somewhat less successful that CVC3; we speculate that it rejects problems with certain types of expressions immediately, even though these expressions do not need to be evaluated to solve the problem. Ergo erroneously reported "Invalid" for several of the verification conditions.

None of the solvers available to us were able to handle all of the verification conditions, but, as they did not prove any conditions to be false, their output still increases the confidence in the correctness of the result.

The process revealed one interesting issue: the third line of our constructed specification for Trimming is incorrect, and so could not be proved (when the configuration would cause a divide-by-zero error, the value returned by trimming is subject to one-point trimming rather than being unchanged). This resulted in a proof failure. This had not been detected during the manual proof as it should have been.

[2]These results generated with Why 2.10b, CVC3 1.2.1, Yices 1.0.13, and Ergo 0.7. Timeout was set to 45 seconds on a 2.6GHz Intel Celeron with 1.5Gb of memory.

2.4.1 *Divide-by-zero*

One significant issue was in handling division. When a division $x/(p-q)$ occurs in the code, Why generates a verification condition to check that division-by-zero cannot occur. If p and q are floats, Why generates the verification condition $p - q \neq 0.0$ (i.e., comparison with the floating-point zero). In the smart sensor code, such divisions are usually guarded by a conditional `p-q!=0`; Caduceus / Why translate this as $p - q \neq \mathit{RealOfInt}(0)$, where *RealOfInt* is Why's uninterpreted casting function. This means that the guard in the code is different from the verification condition.

Our first attempt at a precondition to choose the non-divide-by-zero execution path was to write $p \neq q$. This is a *third* way of representing essentially the same condition, but not all of the decision procedures considered them equivalent. Specifically, CVC3 in CVC-Lite mode could handle these issues correctly, while CVC3 in SMT-LIB mode could not.

2.5 Preliminary Results from other code

At the time of writing, we have begun to apply similar techniques to the "frontend" code from the sensor. This code is older than the core code, and has been inherited from devices earlier in the manufacturer's product line. It has a complex structure: its job is to interface to a wide range of sensor types, so it includes several 1000-line case statements.

The most interesting parts of this code apply the per-sensor linarisation tables to the input values. Unlike the linearisation routines, these are mainly $O(1)$ table lookups with interpolation. The input and output values are floating point, but the table contains integers (with fixed scaling factors), and is of course indexed by integers. Our current automatic proof efforts have stalled as a result of the necessary type conversions—although the manual proofs presented no particular problem. None of the decision procedures that we tried were able to correctly handle conversion between numeric types.

3. CONCLUSIONS

3.1 About the approach

We have found that applying formal proof with the verification assisted by Caduceus, Why and CVC3 is very efficient—it is at least as cost effective as manual proof, and significantly quicker than analysis based on a theorem prover; as the cost-per-proof is reduced, the approach scales better than manual proof. Successfully applying the technique, however, still requires some understanding of the proof techniques, so we do not think that automating the proof reduces the skill-level required. As for all proof techniques, the hardest part of applying the approach is creating a suitable formal specification.

An additional advantage of automatic proof was (unexpectedly) demonstrated: a mismatch between the code and the

Table 1: Solver output on "trimming" example

Verification condition	Context	Ergo	CVC3 (SMT)	CVC3 (CVC-lite)	Yices
pointer dereferencing		Valid	unsat	Valid	unsat
postcondition	divide-by-zero	Invalid	(timeout)	Valid	non linear problem
pointer dereferencing		Valid	unsat	Valid	unsat
precondition	$T_u - T_l \neq 0$	Invalid	(timeout)	Valid	unknown
postcondition	two-point	Invalid	unsat	Divide by a PLUS expression	non linear problem
pointer dereferencing		Valid	unsat	Valid	unsat
postcondition	single point	Valid	unsat	Valid	non linear problem
postcondition	trimming off	Valid	unsat	Divide by a PLUS expression	non linear problem

specification which was not discovered during manual proof, but was identified using the new technique.

3.2 About Caduceus

While the Caduceus/Why/CVC3 combination has been very successful for us, there are a number of shortcomings that could make the tools more easily applicable.

3.2.1 More flexibility on the input language

We would like to avoid modifying the source code as far as possible. Techniques from static analysis could be helpful here: it should be possible to determine automatically that the standard headers, the version control string, and the unusued unions had no impact on the function selected for verification.

It is not clear that our other language issues could be automatically resolved, though. A mapping a set of direct memory accesses to a pseudo-array could be provided as an option, but it would probably be too specific for most users.

3.2.2 More guidance for failed specifications

As described above, we built our specifications in small pieces, as we found that it was difficult to trace a failure to prove a proof obligation back to a particular mismatch between the specification and the code. The visualisation provided in Why's graphical interface helps with this to a certain extent, but a more user-friendly tool could, for example, illustrate the code path which a given proof obligation is exercising.

3.3 Future work

We plan to extend the work to cover more of the available smart sensor code, particularly the system frontend. We also anticipate having access to the source code for a newer smart sensor from the same manufacturer which has been subject to a more rigorous development process, and plan to apply similar techniques to it.

We are also interested in the development of Frama-C [1], which seems set to replace several other analysis tools in use at Adelard as well as extending the functionality of Caduceus.

It is unfortunate that, due to the non-disclosure agreements that make this work possible, we cannot make the code and the detailed specifications available to the academic community. We hope to be able to eventually provide anonymised and abstracted sources to serve as a benchmark for future studies.

4. ACKNOWLEDGMENTS

This work was supported by the FEAST projects, funded by British Energy Generation Ltd, British Nuclear Fuels plc, Magnox Electric plc and AWE plc under the Nuclear Research Programme via contracts 40072849/amcd and 40191209. We are grateful to our industrial partners for providing the case study materials. We are also grateful for the helpful comments from the AFM 2008 reviewers.

5. REFERENCES

[1] Framework for modular analysis of C. Website, 2008. http://www.frama-c.cea.fr/.

[2] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[3] R. Bloomfield and D. Sheridan. Applying formal methods at all SILs. In *INUCE Control and Instrumentation conference*, Manchester, Sept. 2007.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[5] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 55–59, New York, NY, USA, 2007. ACM.

[6] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[7] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[8] International Organization for Standardization. *ISO/IEC 9899:1990: Programming languages — C*. 1990.

[9] A. Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

Detecting erroneous assumptions when verifying software using SMT solvers

David R. Cok
Eastman Kodak Company Research Laboratories
1999 Lake Avenue
Rochester, NY 14650 USA
david.cok@kodak.com

ABSTRACT

Modern static software verification systems use automated or interactive theorem provers to check the consistency of implicit or explicit specifications against software implementations. Systems typically allow users to aid the theorem prover by providing explicit logical statements that are assumed to be true at a given point in the program code. However, there is no check of these assumptions; if an assumption is invalid, verification systems may fail to report other specification or implementation errors, giving the user a false sense of security. This is a known source of unsoundness in current verification technology. This paper analyzes the sources of potentially invalid assumptions and provides procedures to perform such vacuity checking, using SMT-produced unsatisfiability checks and unsatisfiable cores.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Software verification, Automated reasoning, vacuity checking, formal verification

Keywords

specification, verification, theorem prover, soundness, vacuity checking, JML, Java Modeling Language, ESC/Java, ESC/Java2, Yices, Spec#

1. INTRODUCTION

The design of verification systems for practical software has improved significantly in the past several years, encouraged by improvements in efficiency of formulation[10, 15], competitions on prover performance[4, 20], and increasing coverage of programming and specification features. However, in the software development process, most verification tasks fail and many specifications are initially incorrect. A good tool set must provide clear and accurate information both about code that does not conform to its specifications and about specifications that may be incomplete or in error.

Some errors are particularly insidious - those that mask other errors by giving the appearance of verification success. Since the theorem provers embedded in verification systems are not all-powerful, specification languages typically allow the specifier to supply unchecked assumptions. Both Spec#[3] and JML[13, 14, 18], for example, have an `assume` statement that can be placed within a method implementation. This feature is well-known to introduce a potential unsoundness[1] into the verification system[16, 12]. Assumptions are supposed to represent true facts that the verification system is not able to prove. But we have a problem if the user states an assumption that is false (or even sometimes false). In propositional and first-order logical systems, any assertion can be proved from a false premise. Hence typical provers in exploratory use today may pronounce a piece of code valid (that is, matching its specifications) if the user has supplied a false assumption. The false assumption is not reported and may mask the presence of other errors.

Checking for such assumptions is well-known as *vacuity checking* in the model checking literature (e.g. [5, 6]) and is being introduced into run-time testing techniques [1]. However, the problem is not generally addressed by automated tools for formal verification, perhaps because user-written specifications have not yet become widespread (see [11] for initial work). Vacuous assumptions are one case of a larger problem addressed by the question of *specification coverage*: do a program's specifications accurately describe all the behaviors needed to ensure a correctly operating system.

We should be clear concerning the degree to which erroneous assumptions can be detected. A verification system can demonstrate the logical consistency or inconsistency of a program's implementation and its specifications, but the system cannot in general state which is correct when they differ. However, circumstances such as dead code, assumptions that are always or too often false, or unimplementable

[1]Here by unsoundness we mean that there may be errors in the code (as measured by the specification or implicit language semantics) that are not reported by the verification system; a reported warning that does not represent an actual error is called an incompleteness. The terms *unsound* and *incomplete* have been applied to missing error reports and erroneous error reports both in the sense defined here and in the reverse sense.[9]

AFM 2008 Princeton, New Jersey

specifications may be dubious enough to warrant a warning to the user. Any automation of these checks will reduce opportunities for unsoundness in the system. Note that assertions may be vacuous as well – always or too often true; in this paper, we will discuss only assumptions.

Previous work on identifying problematic assumptions in static checking is contained in [11] and [7]; there the pragmatic usefulness of automated tools was established with experimental data. We contribute in addition an expanded discussion of the sources of assumptions in static checking, an analysis of problematic assumptions in the context of the block equation form of verification conditions [2], a comparison of various automated algorithms, and an additional technique that uses unsatisfiable cores for vacuity (and relevance) testing.

The following section summarizes some preliminaries familiar to readers having experience with verification systems. Section 3 explains how verification conditions are formed from program text in order to illustrate how false assumptions can cause unsoundness. Section 4 describes the various ways that a user can inadvertently introduce false assumptions, section 5 describes how verification conditions can be adjusted in order to automatically detect and report such assumptions, and section 6 describes how to use minimal unsatisfiable cores to detect them. The final section summarizes our observations.

2. BACKGROUND

A software verification system has the task of analyzing a program to determine whether the implementation of the program is consistent with the stated specifications. A program that does so is *valid*. In this paper we are only concerned with *static* verification, in which the program and specification text are analyzed without being executed. Specifications may be explicitly stated or may be implicit. An example of an implicit specification is that a program will never attempt to access an array element using an out of bounds index. Note that formal static program validity is a useful step, but not the only step, in assuring correctness of a computational system; the specifications themselves may be incorrect and there are other components of a complete system which must also be correct.

A verification system typically translates a method's implementation and specification into a logical statement called a verification condition (VC). The verification system will use a theorem prover to determine, if possible, whether the VC is valid or not. Ideally, the VC is valid if and only if (iff) the method is valid.

Though the concepts are related and the syntax similar, the reader should distinguish among (a) a programming language's statements and expressions, (b) the specification language used, and (c) the logical theory used by the underlying theorem prover.

- The example code in this paper is in the Java programming language, though the discussion applies equally to C# and other languages.
- The specifications are written in the Java Modeling Language (JML)[13, 14]; translation to Spec#[3] (for C# programs) is straightforward. In these examples, the specifications of a method consist of the method's preconditions (**requires** clauses in JML), frame conditions (**modifies** clauses), and postconditions (**ensures** clauses). Preconditions state the logical conditions that must hold in the program state when the method is invoked; they generally include any object invariants (denoted by **invariant** clauses) that are true of all objects of a given class. Frame conditions tell which program variables are allowed to be modified by the method and, by omission, which are guaranteed to be unchanged after the method call. Postconditions state the logical conditions that must hold in the new program state that exists after a method call; the postconditions also usually include any object invariants.
- Logical statements are expressed using conventional first order logic combined with the theory of arithmetic using conventional notation. Note that in programming and specification language examples the $=$ operator is assignment, while in logical statements $=$ denotes equality. The logic includes variables that correspond in name to programming language variables in method implementations, but typically more than one logical variable is associated with a given programming language variable.

The theorem provers used by most static checkers of the type discussed here are SMT (Satisfiability Modulo Theories) solvers (see [20, 21] for links to a number of tools). These solvers combine decision procedures for propositional logic with others appropriate for reasoning about programming languages: uninterpreted functions, linear arithmetic, arrays, and quantification. These tools have improved significantly in recent years. Their main drawback is that quantified formulae are handled heuristically.

3. THE STRUCTURE OF VERIFICATION CONDITIONS

In this section we are concerned to understand the structure of the verification condition. For this purpose we begin with the translation of a method into a logical statement as described in [2], where the details of the following description can be found.

3.1 Basic blocks and passive programs

Following the derivation in [2], a program is decomposed into an equivalent set of basic blocks, each block corresponding to a non-branching section of code. One distinguished block is the *start block*, corresponding to the beginning of the method (including any preconditions). Logical assumptions are added to indicate the conditions under which a particular basic block may be executed. For example, a simple if-then-else statement will result in two basic blocks, corresponding to the *then* and *else* branches of the if-then-else statement. The *then* block has an initial assumption that the branch condition is true; the *else* block has an initial assumption that the branch condition is false. Each basic block also has a list of blocks which may follow it.

Loops are also converted into basic blocks by the introduction of assumptions and assertions corresponding to loop invariants (see section 4.5). The result is a set of basic blocks that form a directed *acyclic* graph, with the edges corresponding to the *follows* relationship.

Finally, the set of blocks is made into a passive program by transforming each block into dynamic single assignment[2]

[2]Dynamic single assignment (e.g. [2] and references therein)

(DSA) form and then converting all assignments into assumptions. Each block then consists of a sequence of `assume` and `assert` statements. The `assume` statements express logical conditions that are presumed true: they correspond to preconditions, assignments, branch tests, as well as user specified assumptions and axioms. The `assert` statements express logical conditions that must be true, given previous assumptions, if the program is valid; they are generated from explicit assertions and implicit requirements: no dereferencing of nulls, out of bounds indices, calling methods whose preconditions are not satisfied, modifying variables that are stated to be unchanged, etc. A verification system is designed to test and report assertion violations. Current systems do not check for problematic assumptions.

3.2 The block equations

The decomposition of the previous subsection results in a set of basic blocks. Each (basic) block b has the following:

- a sequence $\mathcal{S}_b$ of `assume` and `assert` statements;
- a set F_b of blocks that follow block b;
- an auxiliary propositional variable A_b;
- a block equation Q_b, where Q_b is the formula $A_b = \mathrm{wp}(\mathcal{S}_b, \bigwedge_{j \in F_b} A_j)$ (with wp defined below).

There is a distinguished block, b_0, that corresponds to the beginning of the program. The *follows* relationship $b \succ b' \equiv b' \in F_b$ is a partial order over the blocks and b_0 is a maximal element of that partial order (it does not follow any other block). For simplicity, we restrict ourselves to methods that have a single entry point, in which case b_0 is *the* maximal element. The expressions in the `assume` and `assert` statements and the block equations contain logical variables that are associated with the programming language variables in the method's source code. The DSA procedure converts a given programming language variable into one or more logical variables. This set of logical variables is augmented by the auxiliary variables A_b. The verification condition is this (as proved in [2]), implicitly universally quantified over all the logical and auxiliary variables:

$$(\bigwedge_b Q_b) \Rightarrow A_{b_0}. \tag{1}$$

The interpretation of this verification condition is that the method under scrutiny is valid iff its corresponding VC is true for every assignment of values to the set of logical and auxiliary variables.

Because the blocks can be ordered by the $\succ$ partial order, we can eliminate in turn each auxiliary variable (assuming all the block equations hold) and can solve for A_{b_0}. In order to visualize the structure of that solution, we need to look at the details of the block equation and the definition of the weakest-precondition predicate. The wp predicate operates on its arguments as follows (P, Q, and R are expressions in the underlying logic):

- $\mathrm{wp}(\mathcal{S}, P) = \mathrm{wp}(\mathcal{S}', \mathrm{wp}(T, P))$ if $\mathcal{S}$ is the sequence of statements consisting of the statement sequence $\mathcal{S}'$ followed by the statement T;
- $\mathrm{wp}(\mathcal{S}, P \wedge Q) = \mathrm{wp}(\mathcal{S}, P) \wedge \mathrm{wp}(\mathcal{S}, Q)$ for a statement sequence $\mathcal{S}$;
- $\mathrm{wp}(S, Q) = (P \Rightarrow Q)$ if S is the statement `assume` P;
- $\mathrm{wp}(S, Q) = (P \wedge Q)$ if S is the statement `assert` P.

Using propositional tautologies, we can then express A_{b_0} (or any block variable) in the form

$$A_{b_0} = \bigwedge_m ((\bigwedge_k R_{mk}) \Rightarrow T_m), \tag{2}$$

in which the R_{mk} are assumption predicates and the T_m are assertion predicates. Call each conjunct $(\bigwedge_k R_{mk}) \Rightarrow T_m$ a *trace*. Eq. (2) expresses the VC as a conjunction of traces, with each trace expressing the truth of a given assertion, under a set of assumptions; each trace corresponds to a control path through the program ending at a particular assertion.

Using the transformations given above we can prove (with suitable definitions of each R_{mk} and T_m) that

$$(\bigwedge_b Q_b) \equiv (A_{b_0} = \bigwedge_m ((\bigwedge_k R_{mk}) \Rightarrow T_m)), \tag{3}$$

Thus, an equivalent form of the verification condition is

$$(A_{b_0} = \bigwedge_m ((\bigwedge_k R_{mk}) \Rightarrow T_m)) \Rightarrow A_{b_0}. \tag{4}$$

or, equivalently,

$$\bigwedge_m ((\bigwedge_k R_{mk}) \Rightarrow T_m), \tag{5}$$

which we abbreviate as A_{VC}.

The method under scrutiny is valid precisely when its A_{VC} is true. A_{VC} is true precisely when all of its traces are jointly true for any given assignment. A trace can be false for a given assignment only when its assertion is false and all of its assumptions are true; if any assumption evaluates to false, the trace will be true. As is explained shortly, an assumption can be false for very legitimate reasons. The crux of the problem addressed by this paper is found in assumptions that are unexpectedly false but are not distinguished from assumptions that are legitimately false.

The expanded form of the VC in Eq. (5) is unsuitable for input to a theorem prover. For any program with branches and loops, this VC will be larger than it would be by supplying the theorem prover with the basic block equations (Eq. (1)) directly. Distributing $\Rightarrow$ over $\wedge$ introduces copies of common subexpressions that will cause a theorem prover to do unnecessary work. The form of Eq. (5) however is conceptually simple and allows easier reasoning about the properties of the VC.

3.3 Assumptions produced by passifying assignments

An important step in producing a verification condition from a program is renaming variables into DSA form and then passifying assignments by expressing them as assumptions. For example, the program text

```
x = 1;
x = x + 1;
assert x == 2;
```

becomes the passified program

```
assume x$1 = 1;
assume x$2 = x$1 + 1;
assert x$2 = 2;
```

is similar to the better known static single assignment, but that distinction is not used in the discussion in this paper.

which becomes the VC fragment

$$((x\$1 = 1) \wedge (x\$2 = (x\$1 + 1))) => (x\$2 = 2).$$

In considering satisfying assignments, $x\$1$ and $x\$2$ are assigned (numeric) values independently. There are many variable assignments in which $x\$1$ is different than 1 and many variable assignments in which $x\$2$ is different than $x\$1 + 1$. All of these "incorrect" variable assignments result in false assumptions and a true VC. In this case, only one variable assignment satisfies the two assumptions ($x\$1 = 1$ and $x\$2 = 2$) and with that assignment the final assertion is true.

Thus assumptions that are false under some assignments are an integral and expected part of the design of a VC. The problem is that false assumptions can also be introduced inadvertently, as described in section 4.

3.4 Feasible assumptions and traces

It is helpful to define the concepts of (in)feasible assumptions and traces, in analogy to feasible paths through sequences of programming language statements.

- An assumption R_{mk} is *immaterial* within a trace $(\bigwedge_k R_{mk}) \Rightarrow T_m$ iff $(\bigwedge_{j<k} R_{mj})$ is false for all variable assignments.
- An assumption R_{mk} is *infeasible* within a trace $(\bigwedge_k R_{mk}) \Rightarrow T_m$ iff for all variable assignments for which $(\bigwedge_{j<k} R_{mj})$ is true, R_{mk} is false.
- An assumption is *strictly infeasible* if it is infeasible but not immaterial.
- An assumption R_{mk} is *feasible* within a trace iff it is not infeasible within that trace, that is, an assumption is feasible iff there is at least one variable assignment for which $(\bigwedge_{j<k} R_{mj})$ and R_{mk} are true.
- A trace is *infeasible* iff it contains at least one infeasible assumption.
- A trace is *feasible* iff it is not infeasible, that is, none of its assumptions are infeasible.

The following observations follow easily from these definitions.

- An immaterial assumption is infeasible, but not strictly infeasible.
- If an assumption R_{mk} is immaterial, then some R_{mj} for $j < k$ is infeasible.
- An assumption is (exclusively) either strictly infeasible, immaterial, or feasible.
- An infeasible trace is always true (even if the concluding assertion is always false).
- If a program contains a feasible trace whose final assertion is `false`, then there is a variable assignment for which the trace is false and the program is therefore not valid.
- An infeasible assumption is not necessarily false for all variable assignments, but it is always false when its preceding assumptions are true, that is, it is always false when it is "relevant".

4. SOURCES OF FALSE ASSUMPTIONS IN USER SPECIFICATIONS

Assumptions in passive programs are produced by the following programming language and specification constructs:

- assignment statements
- branch conditions (e.g. if-then-else and switch statements and loop tests)
- loop invariants
- method preconditions
- specifications of called methods
- user-specified assume statements (including JML's axiom clause)

The assumptions from assignment statements were discussed in subsection 3.3. From that discussion and the definitions in subsection 3.4, assumptions from assignments are never strictly infeasible. Branch and loop conditions are expected to be feasible (if not immaterial), but can be associated with dead code (subsection 4.6) and infinite or unexecuted loops. Loop invariants are discussed in subsection 4.5.

The last three of the sources listed above are user-written specifications that can introduce unintentionally false assumptions. Each of these false assumptions stems from a *user* error; that is, the user has written an incorrect specification, and the theorem prover is giving logically correct conclusions from the faulty premise.

4.1 Faulty assume statement

The most obvious and significant source of a faulty assumption is a user-supplied `assume` statement. The condition contained in the `assume` statement may be tautologically false - a misstated mathematical axiom, for example. Less obvious is an assumption that is always false on some branch of the program's control flow, but not necessarily false for all assignments of program variables.

Consider the example[3] code in Fig. 1. The passified program has these statements:

```
assume i$0 > 0; // precondition
assume j$1 = 0; // assignment
assume i$0 = 0; // assumption
assert j$1 > 0; // assertion
```

The VC is

$$((i\$0 > 0) \wedge (j\$1 = 0) \wedge (i\$0 = 0)) \Rightarrow (j\$1 > 0).$$

The third assumption contradicts the first and is infeasible, so this VC is always true. As a result, the erroneous assertion is not reported.

4.2 Faulty preconditions

The preconditions of a method are encoded in the verification condition as assumptions that begin the verification condition or the starting block equation. If any combination of the preconditions is always false, the VC will be trivially true because of an infeasible assumption. Note that the preconditions include both the explicit preconditions of the method being verified as well as class invariants, which are typically separated textually from the method specifications.

Consider the code in Fig. 2. A method `fix` is supposed to change a static variable from negative to positive, so that the class invariant is satisfied, but has a typo. The passified basic block for this method has the following statements:

[3]The examples in this paper are intentionally obvious and simple, to save space and to make the illustrations clear. Specification errors can be obvious in small programs, but still be hard to spot in large real systems. Practicing writers of software and specifications will be able to remember their own more complex examples.

```
class E {
    //@ requires i > 0;
    static void problem(int i) {
      int j;
      j = 0;
      assume i == 0; // should be j == 0
      assert j > 0;  // should be i>0
    }
}
```

Figure 1: Example code with an erroneous assumption that hides an invalid assertion

```
class E {
    static int i;
    //@ static invariant i > 0;

    //@ requires i < 0;
    //@ modifies i;
    //@ ensures i > 0;
    static void fix() {
      i = -1; // should be i = -i
    }
}
```

Figure 2: Example code with an infeasible precondition

```
assume i$0 > 0; // invariant
assume i$0 < 0; // precondition
assume i$1 = -1; // assignment
assert i$1 > 0; // postcondition
```

The VC for this method is then

$$((i\$0 > 0) \wedge (i\$0 < 0) \wedge (i\$1 = -1)) \Rightarrow (i\$1 > 0).$$

This VC is always satisfied because the first two assumptions are contradictory; consequently no warnings are given, even though the postcondition assertion should not hold. The specifier apparently forgot that the class invariant is part of the precondition as well. If the invariant is removed, then the failure to satisfy the postcondition is noted.

4.3 Faulty postconditions of called methods

If the body of a method M calls a method M', then the specifications of method M' must be considered in the verification of method M. The specification contract of method M' states that if M' is called in a state in which its preconditions hold, then upon completion its postconditions will hold. A called method is encoded by (1) assigning values to the formal parameters, (2) introducing assertions for its precondition, (3) defining new variable names for modified program variables, and (4) adding assumptions expressing its postconditions. Whether or not the postconditions of M' are actually established by its implementation, given its preconditions, is determined when M' is verified; during the verification of M that relationship is assumed. However, M' may be undergoing debugging itself and its specifications are still in error, or it may be part of a library for which verification has not (yet, we'll say) been performed. Although we cannot guard against all specification errors, we can check for infeasible postconditions, that is, for post-

```
class E {
    static int i,j;

    static public void caller() {
      j = 0;
      i = sign(j);
      //@ assert i == 1000; // Should fail
    }

    //@ ensures (k >= 0) ==> \result == 1;
    //@ ensures (k <= 0) ==> \result == -1;
    static public int sign(int k);
}
```

Figure 3: Example code with an infeasible postcondition

conditions that are contradictory (unimplementable).

Consider the example code in Fig. 3. The statements of the basic block corresponding to the method `caller` are these:

```
assume j$1 = 0; // the assignment
assume k$1 = j$1; // the formal parameter
assume (k$1 >= 0) ==> ($result$1 = 1);
assume (k$1 <= 0) ==> ($result$1 = -1);
assume (i$1 = $result$1); // assignment
assert (i$1 = 1000); // the assertion
```

This becomes the verification condition fragment

$$\begin{aligned}((j\$1 = 0) \quad &\wedge \quad (k\$1 = j\$1) \\ &\wedge \quad ((k\$1 \geq 0) \Rightarrow (\$result\$1 = 1)) \\ &\wedge \quad ((k\$1 \leq 0) \Rightarrow (\$result\$1 = -1)) \\ &\wedge \quad (i\$1 = \$result\$1)) \\ \Rightarrow (i\$1 &= 1000).\end{aligned}$$

Any variable assignment that makes the first three assumptions true must have $j\$1$ and $k\$1$ be 0 and $\$result\$1$ be 1. With any such assignment, the fourth assumption will always be false. Hence the fourth assumption is infeasible, the trace is true, and the false assertion generates no warning.

Note first that the specification of `sign(int i)` is incorrect; that error would be found when trying to verify the `sign` method. However, it would be helpful to the programmer trying to specify `caller` to provide as much relevant information as possible.

Also, note that this detection of infeasible postconditions in context is weaker than verifying the called method directly. If, in `caller`, the first assignment were `j = 1;`, then no infeasible assumptions would be found, even though the postconditions are not entirely correct; they do work fine if the argument to the method is not 0.

4.4 Missing frame conditions

In the previous section, we talked only about the pre- and postconditions of a called method. There can also be errors in its frame conditions that lead to infeasible assumptions. Consider the example in Fig. 4. The specifications of method `badinc` have an incorrect frame condition (`modifies` clause): it is missing the fact that variable `i` is changed in the method. The DSA method of creating a verification condition uses a new variable name in the VC to represent

```
class E {
    static int i,j;

    //@ modifies i;
    //@ ensures i = j + 1;
    static public void inc() {
      i = j+1;
    }

    //@ modifies \nothing; // Incorrect
    //@ ensures i == j + 1;
    static public void badinc() {
      i = j+1;
    }

    static public void caller() {
      ...
      i = 0;               // Line A
      //@ assume j == 0;   // Line B
      inc();               // Line C
      //@ assert i == 1;   // Line D
      j = i;               // Line E
      badinc();            // Line F
      //@ assert i == 2;   // Line G
      ...
    }
}
```

Figure 4: Example code with an incorrect frame condition

a program variable each time that program variable might be modified; program variables can be changed by direct assignment or indirectly in method calls. The call to `inc()` results in the following statements:

```
assume i$1 = 0;       // Line A
assume j$1 = 0;       // Line B
assume i$2 = j$1 + 1; // Line C
assert i$2 = 1;       // Line D
```

Note that on Line C, after the call of `inc()`, a new variable, $i\$2$, is used to represent the program variable `i`. This variable is introduced because the specification of `inc` indicates that `i` may be modified. The assertion for Line D is consistent with the assumptions on Lines A–C. On the other hand, the call of `badinc()` will result in the following statements:

```
assert i$2 == 1;       // Line D
assume j$2 == i$2;     // Line E
assume i$2 == j$2 + 1; // Line F
assert i$2 == 1000;    // Line G - should fail
```

Here, the method `badinc` specifies that `i` is not modified in the method call; therefore, the same variable name is used for the program variable `i` before and after the method call. Because `i` does not get a new variable name in Line F, the assumption on Line F contradicts the assumption on Line E and consequently is infeasible. No warning is given for the false assertion on Line G.

4.5 Faulty loop invariants

Loop invariants also introduce assumptions. However, faulty loop invariants generally result in unprovable assertions rather than silent infeasible assumptions. As explained in [2], loops are decomposed into an acyclic graph of basic blocks by introducing appropriate assumptions and assertions. A loop invariant is expected to be true prior to each iteration of a loop. A loop of the following form

```
...
//@ loop_invariant I;
while (B) {
   ...
}
...
```

results in basic blocks like these:

```
Before:
   ...
   assert I ; // asserts the loop invariant
   goto Loop

Loop:
   assume I'; // loop invariant with renaming
   goto Body, After

Body:
   assume B ; // loop condition is true
   ...
   assert I'';// loop invariant with renaming
   goto       // goes nowhere

After:
   assume !B ;// loop condition is false
   ...
```

If `I` is tautologically false, the assumption in the Loop block will always be infeasible. However, the assertion in the Before block will also fail, producing a warning to the programmer. In the case where the loop invariant is not tautologically false but is faulty because it contradicts assumptions earlier in the program, the assertion will still fail. The situations in which `I` combined with `B` or `!B` results in dead code correspond to situations in which the loop never terminates or is never executed; these can be detected as described in the following subsection. So no special checks are needed for loop invariants.

4.6 Dead code

Consider the code of Fig. 5. The then part of the if statement is never executed, presuming any caller does indeed satisfy the method's preconditions. The basic blocks for this snippet of code contain the following:

```
Start:  assume o != null;  // A
        goto Then, Else

Then:   assume o == null;  // X
        ...

Else:   assume o != null;
        ...
```

The assumption marked X is infeasible because it is always preceded by the assumption marked A. Infeasible assumptions resulting from never-taken conditional branches do not cause the verification system to give misleading results. However, they do indicate dead code of which the programmer may be unaware. The same mechanism that

```
class E {
     static int i;

    //@ requires o != null;
    static public void example(String o) {
      if (o == null) {
          i = -1;
      } else {
          i = o.length();
      }
    }
}
```

Figure 5: Example dead code

discovers other kinds of infeasible assumptions can find dead code as well and can provide at least the option of warning the programmer.

If there are multiple branches within a program, it is common that, even if no branch is dead, various combinations of branches cannot be executed together. For example, if a program has many if statements with the same branch condition (e.g. `if (DEBUG) ...`), then all of those branches will be executed or none of them will. Thus there may be many combinations of branches (i.e. many traces) that are correctly infeasible. They will be apparent because they will have contradictory assumptions arising from branch conditions (e.g. both `assume DEBUG` and `assume` $\neg$`DEBUG`). Any method to warn the programmer of infeasible assumptions will need to avoid issuing false warnings in such cases.

5. DETECTING FAULTY ASSUMPTIONS

5.1 Testing the expanded VC

So far we have observed that VCs contain a mixture of assumptions, some of which are legitimately false for a given variable assignment, others are appropriately infeasible in a given trace, and others may be infeasible because of user errors. In this section we describe methods to detect infeasible assumptions using satisfiability checking of SMT provers. We will check for the following potentially faulty specifications by testing whether the corresponding assumptions introduced into the VC are infeasible:

- faulty explicit user assumptions;
- faulty preconditions;
- faulty frame and postconditions of called methods
- code blocks that are never executed (dead), by testing for infeasible branch and loop conditions

In the process we must avoid warning about

- assumptions generated by assignment statements
- infeasible assumptions (stemming from branch conditions) on legitimately infeasible traces, so long as the basic block controlled by the branch condition is part of some feasible trace

As noted above, if a particular trace of a VC contains an infeasible assumption, that trace will always be true, even if the assertion statement is explicitly false. This fact corresponds to an idiom employed by ESC/Java2[8] users: if the user suspects that ESC/Java2 reports no errors for a particular method because of an infeasible assumption, then an `assert false` statement is added at an appropriate place in the method's code; if ESC/Java2 now reports an assertion violation, then the assumptions (at least up to the location of the inserted assertion) were feasible; however, if no assertion violation is reported, then there is something infeasible preceding that assertion.

Note that there is no point in checking for infeasible assumptions while the VC has assertion violations. In fact, if an assertion is identified as false in a particular trace, then that trace must be feasible. It is only when no assertion violations are reported that we want to check that all assumptions are feasible, in order to know whether to trust the no-assertion-violation report. All that is needed to ensure that an assumption is feasible in a trace is to test that the appropriate $\wedge_{j \leq k} R_{mj}$ is satisfiable.

Thus we can do the following:

Algorithm A:

> For each assumption R_{mk} to be tested in each trace $(\bigwedge_k R_{mk}) \Rightarrow T_m$ of the VC, do the following: Construct the conjunct $\wedge_{j \leq k} R_{mj}$. The conjunct is satisfiable iff R_{mk} is feasible.

Algorithm A tests each copy of each assumption in each trace individually; each test is precise, but many tests are required. There is a trace for each path through the code, and the number of traces is exponential in the number of branch statements (e.g. if-then-else or switch statements). We can reduce the number of tests by checking each trace once, as follows:

Algorithm B:

> For each trace $(\bigwedge_k R_{mk}) \Rightarrow T_m$ of the VC, do the following: Construct the conjunct $(\bigwedge_k R_{mk})$; the trace is feasible iff this conjunct is satisfiable.

Algorithm B allows testing all assumptions in a given trace at once. If the test fails, one must still narrow the search to a particular faulty assumption. However, for the common case of checking that a mostly correct program has not acquired faulty assumptions, this approach reduces the work required. Unfortunately, there still may be too many traces to test. And if many traces are unfeasible because of conflicting branch conditions, a lot of unnecessary work may result.

One cannot, however, combine multiple traces into one test. Suppose we modify Algorithm B to test two traces at once by conjoining two created traces. If the resulting expression is valid, then *both* traces contain an infeasible assumption. If the conjunct is invalid, then one or the other trace is feasible; that is, if the conjunct is invalid, there may be no infeasible assumptions or there may be one. This combined test is not particularly helpful in assuring that there are no infeasible assumptions in the program.

5.2 Testing the basic block VC

There are two problems with the algorithms of the previous subsection: there may be too many traces to check, and we would prefer to perform our validity testing using the much more efficient block equation form (Eq. (1)) of the VC. Observe that when the block equations are expanded into Eq. (5), the predicate of a given assumption may be replicated across multiple traces, corresponding to the various control paths by which it can be reached within the program.

Suppose we execute the following procedure:
Algorithm C: For each `assume` statement that is to be tested in the set of basic blocks, do the following:

- place a `assert false;` statement immediately after that assumption
- regenerate the block equations and the modified VC
- if the modified VC is still valid, the assumption is infeasible in all the traces in which it is present; if the modified VC is invalid, the assumption is feasible in some but not necessarily all traces in which it is present.

Alternatively (**Algorithm C'**), we can combine checks for all assumptions in a basic block by adding the assertion at the location after all the assumptions in the basic block.

5.2.1 Dead code

This test is precisely what is needed for dead code detection. Assumptions from branch conditions may be feasible or infeasible depending on the control path taken to reach the basic block corresponding to the branch. We do not wish to report branches that are infeasible in some traces, only those which are infeasible in all traces in which they occur. Algorithm C or C' does precisely that.

5.2.2 Preconditions

This test is also trivially appropriate for testing preconditions. A method's preconditions occur as assumptions at the beginning of the start block. Those assumptions form the first portion of every trace. Hence they will be infeasible in all traces or in no traces. Algorithm C will correctly indicate whether the precondition assumptions are infeasible (contradictory). One can also, equivalently, and perhaps more easily, test that the conjunction of the precondition assumptions is satisfiable, indicating they are jointly feasible.

5.2.3 User assumptions

When Algorithm C is applied to explicit user assumptions, it tests whether the assumption is infeasible in all traces (on all control paths). This is a good first test to apply and checks, for example, that the assumption is not always false. But it is also useful to be able to test that the assumption is not strictly infeasible on individual traces.

We can test the feasibility of a block B (or an assumption in that block) in a particular trace T in the style of Algorithm C as follows. Determine the set of blocks $\mathcal{B}_B$ that precede B in T. Let P_b be the branch condition (if any) corresponding to a block b in $\mathcal{B}_B$. The conjunction $(\wedge_{b\in\mathcal{B}_B} P_b)$ is true in block B precisely for the specific trace of interest. Thus we have this algorithm:
Algorithm D: For each `assume` statement that is to be tested in a block B (or for the entire block), do the following:

- form the negated conjunction $\neg((\wedge_{b\in\mathcal{B}_B} P_b))$
- place a `assert` $\neg((\wedge_{b\in\mathcal{B}_B} P_b))$`;` statement immediately after that assumption or at the end of the block
- regenerate the block equations and the modified VC
- the assumption or block is infeasible in all the traces for which the assertion is false iff the modified VC is still valid.

The advantage of this algorithm is that the predicate in the assertion can be chosen to designate any specific trace or group of traces; the entire VC does not have to be rewritten – only the one subformula.

5.2.4 Specifications of called methods

We can treat called methods in precisely the same way as user-specified assumptions.

5.3 Observations

The sections above described two basic procedures for detecting infeasible assumptions using satisfiability checking:

- Algorithms A,B,D: Check that each trace is feasible. If not, verify that the trace is infeasible because of a user assumption, rather than a combination of branch conditions.
- Algorithm C,C': Check that each assumption or block in the block equations is feasible (for at least one trace).

The choices among these depend on the performance characteristics and features of the prover being used. When invocations of the prover are expensive we would start with Algorithm C', and then use Algorithm D when more precision about assumptions being feasible in each potential trace are needed. If the prover's performance scales badly with the size of the logical statement under scrutiny, then Algorithm B (or A) has an advantage, even though it may need repeating for many traces.

If the prover being used provides incremental satisfiability results efficiently, that feature can be used to advantage. For example, Algorithm D can be modified as follows. Rather than inserting the statement, `assert` $\neg((\wedge_{b\in\mathcal{B}_B} P_b))$`;`, insert instead the statement `assert X;`, where `X` is a previously unused variable. In addition add the additional equation (like a block equation) $X = \neg((\wedge_{b\in\mathcal{B}_B} P_b))$. In an incremental SMT prover, that latter equation can be retracted and an alternate equation (for a different trace) asserted, and satisfiability rechecked. Depending on the prover, this may be a more efficient procedure than repeated satisfiability checks from scratch.

It is also useful to factor a VC using definitions. If there is a common subformula or subterm T, replace it everywhere by an otherwise unused variable Z and conjoin the equation $Z = T$ to the VC. This can shorten the VC, simplify the prover's task, and assist in interpreting any counterexample (particularly if Z is appropriately named). It can also simplify the task of formulating various additional logical statements to be checked.

There is a final observation to make. With user-specified assumptions, we are concerned to know that the assumption is feasible in any trace containing that assumption. This can add significantly to the testing time. However, if the assumption is independent of the control path used to reach it, only one test need be performed. This will be the case if the assumption is written as a fully-quantified tautology with no free variables, presumably one that the prover cannot prove itself; the tautology would equate logical assertions known to the prover prior to the assumption to those needed to check the remainder of the program.

6. USING UNSATISFIABLE CORES

The previous section outlined automated methods to find infeasible assumptions using a prover's unsatisfiability check. Some SMT provers provide additional information when a set of formulae are unsatisfiable: a subset of the formulae, called an *unsatisfiable core*, which is itself unsatisfiable. This information can be used to identify vacuous or irrelevant

information, though, as we shall see, some improvements in the state of the art of SMT solvers are needed to make this a robust technique. It also requires formulating the VC differently than above. Checking of unsatisfiable cores is being used in model checking tools but has not yet been applied to static checkers using SMT provers.

Consider this simple basic program: `assume X; assume Y; assert X;`. To verify this, one establishes the unsatisfiability of the set $\{\neg A, A = (X \implies Y \implies X)\}$. Both formulae are needed to show unsatisfiability. This is still true if Y is replaced by $\neg X$. However, let us factor out the assertion. The set $\{\neg A, A = (X \implies \neg X \implies T), T = X\}$ is unsatisfiable, but the first two formulae constitute an unsatisfiable core; the third formula is not needed.

We can apply this approach in general as follows. In checking a VC for satisfiability, we rewrite the negation of the VC in conjunctive normal form as $\neg A_{b_0} \wedge (\wedge_b Q_b)$; then each conjunct becomes a formula that may or may not be present in the unsatisfiable core. The VCs produced from block equations are especially amenable to this transformation, as opposed, for example, to those produced by the current version of ESC/Java. We modify the original block program by inserting an `assert Z;` statement (where Z is a previously unused variable name) immediately after an assumption S or at the end of a block that we wish to check, and from that we generate a new verification condition, VC'. Then we can note these two facts:

- $\neg VC$ is unsatisfiable iff $((\neg VC') \wedge Z)$ is unsatisfiable;
- $(\neg VC')$ is unsatisfiable iff $((\neg VC') \wedge Z)$ is unsatisfiable and the assumption S is infeasible (on all paths to S).

Therefore, if $\neg VC$ is unsatisfiable and Z is not a member of the minimal unsatisfiable core, then S is infeasible.

Intuitively, if we place a `assert Z` in a program, then (a) if Z is true, the program is unchanged, but (b) if the value of Z does not matter, then execution never reaches that point in the program.

We can also extend this technique analogously to Algorithm D by introducing a guard that constrains the check to a particular trace (or group of traces). Instead of the simple assertion Z, we place the assertion $\neg G \vee Z$ in the location to be tested, where G is a formula that is true only for the traces to be tested. Then the assertion is true for all other traces, and if the trace is feasible up to the assertion, then Z will be needed in the minimal core; if the trace is infeasible, Z will not be in the minimal core.

Though this technique works in principle (and partially in practice), there are a few issues. First, a *minimal* unsatisfiable core is needed and producing a minimal unsatisfiable core is difficult. For example, Yices[17, 22] produces unsatisfiable cores, but does not guarantee that they are even locally minimal. Without a guaranteee of minimality, the procedure above is not robust by itself; one needs in addition to check and correct the reported core for minimality.

Second, the core produced can be sensitive to the way in which the set of formulae is presented to the prover. For example, if a common subformula is factored out and defined in a separate equation, then an additional formula is needed as part of the minimal core, changing perhaps what is minimal. It could be useful to measure minimality not simply by a count of formulae, but by some weight attached to each formula.

Finally, the overall performance cost of determining unsatisfiable cores on real programs has not been measured. For this algorithm to be useful it must be possible to use cores and incremental satisfiability checks in a way that improves upon the procedures in section 5. A related question is whether incremental checks using a SMT prover's ability to save, restore or incrementally change its logical context improves the performance of doing multiple satisfiability checks over that of restarting the prover (as is done, for example, in [11]).

7. IMPLEMENTATION

A prototype implementation sufficient for experimentation has been created using the OpenJDK[19] implementation of a Java compiler (for Java 7). This compiler was extended to parse all of JML into an internal abstract syntax tree and to perform syntax and typechecking on most of JML's features. The Java/JML AST for a given method is translated into verification conditions appropriate for input to the Yices[17, 22] SMT solver using a partial, proof of concept implementation. Techniques of both sections 5 and 6 were implemented.

The ability of Yices to accept and retract assertions incrementally is helpful here. Once a method's VC is found valid (no assertion violations), the verification system can then assert and retract the additional logical statements that represent the checks for various infeasible assumptions. This incremental approach avoids having to restart the prover and reassert the entire background predicate and block equations for the method under test. Similarly, the unsatisfiable cores produced by Yices are sufficient to experiment with the procedure of section 6.

The implementation to date has allowed experimentation with the concepts and procedures described in this paper. Assessment of their performance on significant code bases, the evaluation of various heuristics in formulating the VCs, and comparison of performance across various SMT solvers is in progress.

8. RELEVANCE

We note as future work that the techniques of section 6 can be used to measure relevance and coverage as well as to check vacuity. Suppose we replace some subterm T in the block equations with a new variable Z, and add to the set of block equations the definition $Z = T$. The satisfiability of the resulting VC is unchanged. However, if the VC is unsatisfiable and the formula $Z = T$ is not part of the unsatisfiable core, then the value of T is irrelevant to the proof of validity supplied by the unsatisfiability check. That is, that portion of the program is not relevant to the specification being checked. Note that T may represent a group of statements in a program, or it may be a small subterm within an expression. Such irrelevance can indicate a fault on either side.

- On the one hand, that portion of the program may actually not be needed. It may be never executed code, but it may also be something executed but not used, such as an unused variable assignment.
- On the other hand, it may be that the specification is not detailed enough, that is, the *coverage* of the specification is inadequate (just as a runtime test set might have inadequate coverage). In some cases, of course, the user may not want or need a completely

detailed specification, but in others this condition may point to inappropriate underspecification.

We can similarly use this technique to perform the complementary check that a subterm of the specification is relevant. If it is not relevant, it is likely that some portion of the specification is vacuous.

9. CONCLUSIONS

We have observed in this paper that some kinds of erroneous, user-supplied specifications can be detected automatically by an enhanced software verification system and have proposed procedures for doing so. This reduces (but does not eliminate) the risk of misunderstood results from SMT-style provers caused by faulty assumptions. The enhanced verification system can use the components (e.g. parsers, VC generators, provers) that are in common use today for such systems; it only need check some additional logical assertions that are related to the primary VC for a method. The incremental checking capabilities of some SMT solvers are useful in this process. In addition, if the underlying prover is able to report unsatisfiable core sets of assertions, then some information about infeasible (vacuous) assumptions can be obtained without repeatedly invoking the prover.

10. REFERENCES

[1] T. Ball and O. Kupferman. Vacuity in testing. In *Proc. 2nd International Conference on Tests and Proofs*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In M. D. Ernst and T. P. Jensen, editors, *Program Analysis For Software Tools and Engineering (PASTE)*. ACM, Sept. 2005.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[4] C. Barrett, L. de Moura, and A. Stump. Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005). *Journal of Automated Reasoning*, 35(4):373–390, 2005.

[5] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 596–602, New York, NY, USA, 1994. ACM.

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290, London, UK, 1997. Springer-Verlag.

[7] P. Chalin. Early detection of JML specification errors using ESC/Java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 25–32, New York, NY, USA, 2006. ACM.

[8] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[9] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, July 2006.

[10] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, 2001.

[11] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 23–30, New York, NY, USA, 2007. ACM.

[12] J. Kiniry, A. E. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006)*, pages 19–24. ACM, Nov. 2006.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[14] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.

[15] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6), Mar. 2005.

[16] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, Oct. 2000.

[17] J. Rushby. Tutorial: Automated Formal Methods with PVS, SAL, and Yices. *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, 11-15 Sept. 2006.

[18] Many papers regarding JML can be found on the JML web site: `http://www.jmlspecs.org`.

[19] The OpenJDK web site hosts the code and other documentation: `http://openjdk.java.net`.

[20] The SMT-COMP web site provides results of the SMT competition and links to the system descriptions of the participants: `http://smtcomp.org`.

[21] The SMTLIB web site hosts benchmarks for SMT solvers and defines a common SMT input language: `http://combination.cs.uiowa.edu/smtlib`.

[22] The Yices web site provides documentation and technical reports: `http://yices.csl.sri.com`.

Strengthened State Transitions for Invariant Verification in Practical Depth-Induction*

Péter Bokor†
Budapest University of Technology
Budapest, Hungary
petbokor@mit.bme.hu

Sandeep Shukla
Virginia Tech
Blacksburg, USA
shukla@vt.edu

András Pataricza
Budapest University of Technology
Budapest, Hungary
pataric@mit.bme.hu

Neeraj Suri
Technische Universität Darmstadt
Darmstadt, Germany
suri@cs.tu-darmstadt.de

ABSTRACT

Bounded Model Checking (BMC) is often able to handle thousands of system variables by encoding the system and its properties via symbolic formulas and using satisfiability (SAT) solvers for verification. To further ease the verification of state invariants, BMC is augmented with a general induction rule called k-induction; however, this sacrifices completeness. Invariant strengthening, a method proposed to overcome this problem, often requires user intervention which limits its general applicability.

This paper presents a systematic method which is able to prove every property that is provable with standard k-induction and, in addition, further properties that the standard technique is unable to prove might be provable as well. Our case studies demonstrate the benefit of our approach with respect to plain k-induction. The main idea is to constrain the state transition relation in a way that the space of reachable states remains unchanged and k-induction is more likely to succeed. We show an implementation of our technique where the user needs only to extend the guard conditions with invariants obtained from the system's specification. This is always possible if the schedule of the executed transitions is (partially) known a-priori.

1. INTRODUCTION

Model checking [6] has proven to be one of the most effective verification techniques. Over the last two decades, model checking has been applied to various fields including hardware, software and protocol verification. The growing demand for verifying large systems has made model checking of explicit state models infeasible, therefore, the system and its properties were proposed to be represented by symbolic formulas. In fact, Binary Decision Diagrams made it possible to verify systems with hundreds of variables [10], SAT-based methods have gone even beyond that [2]. The latter approach is called *Bounded Model Checking* because only execution paths of limited length are translated into input formulas of the SAT engine. The technique guarantees that, given a bound k, the SAT instance is satisfiable if and only if there exists a counterexample (i.e., a violating execution) whose maximum depth is k. The limitation of the BMC approach is that general completeness, i.e., the guarantee that correct properties can always be proven, comes at high price. The intuition is to find the diameter of the system which can be thought of as the longest depth to exhibit any counterexample. In practice, however, this value is often too large for effective analysis.

To mitigate the "depth-explosion" of BMC for the verification of state invariants, an adaptation of mathematical induction was proposed in which, under favorable circumstances, no exhaustive exploration of the state space is needed [17]. Despite the practical success of k-induction, the application of this approach is limited by the fact that the proof of valid invariants might fail due to *spurious counterexamples*. Such runs are false negatives because they lead to unreachable states and are thus invalid executions of the system. A possible solution is to increase the induction depth, which, however, does not scale since the best known SAT algorithms are exponential in the number of input variables. This solution, even if the complexity of SAT was manageable, might not work for systems with infinitely long paths if a spurious counterexample exists for any large k. As an alternative solution, *invariant strengthening* has been proposed which constrains the input formula of the SAT solver such that no spurious counterexamples are generated and soundness of the algorithm is maintained (e.g., [9]). *The contributions of this paper are to (1) present a new method of invariant strengthening, which can be used in conjunction with other*

*Research supported in part by DFG GRK 1362 (TUD GK MM), EC Genesys and ReSIST.

†Péter Bokor is also with Technische Universität Darmstadt.

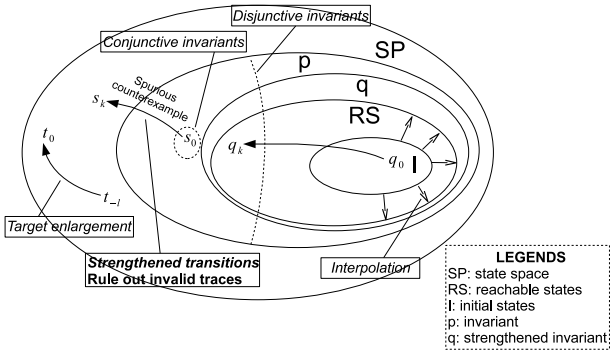

Figure 1: Comparing strengthened transitions with other methods

existing techniques, to (2) show an implementation of the proposed general technique, and to (3) demonstrate its practical usability in protocol verification.

1.1 Positioning the Method and Related Work

The verification of a safety property p is to decide whether every reachable state satisfies p. The different methods of induction-based safety verification can be categorized based on how the verification problem is (re-)formulated (Figure 1).

Assume that k-induction returns a counterexample which starts in the non-reachable state s_0 and reaches s_k after k steps such that the goal invariant (denoted by p) is violated in s_k (see Figure 1). This is a spurious counterexample because s_k is not reachable. If such a run exists, we say the property p is non-inductive. As a general solution, invariant strengthening was proposed where q (the strengthened invariant) is assumed to be inductive and p can be proven if q implies p. The question is how to obtain q? The different approaches can be classified as bottom-up or top-down. The first class only considers the given program, the approaches in the second class are guided by the goal assertion p. System-independent solutions for both classes are given in [13]. Specific knowledge about the system can also be used to find the appropriate q (e.g., see the study about the analysis of protocols for low-level data transmission [5]). Another systematic approach, called conjunctive invariants [9], uses the spurious counterexample to rule out s_0 as a possible state and derives the strengthened invariant $q = p \wedge \neg s_0$. The execution of k-induction can be preceded by a BDD-based pre-image computation of the undesired states (target enlargement) [1]. The predicate describing the states that can reach the target t_0 in l steps can be used to strengthen the invariant[1]. Another form of strengthened invariants is to prove the invariants of auxiliary formulas (defined as a lemma) and discard spurious counterexamples that contradict any of the lemmas.
A different strategy is using disjunctive invariants where the invariant of the system is built incrementally such that it is weakened every time k-induction finds a valid counterexample [15]. For example, q_k is a reachable state from initial state q_0 and the fact that q_k violates p proves that p is not an invariant of the system. Property p is now extended such that all configurations in the set of reachable states are covered. Another approach is using interpolants which are over-approximations of the set of reachable states; approximations are computed based on the unsatisfiability of reaching a violating state after up to k steps [14]. Interpolants are inductively computed (starting with the initial set of states) such that the resulting set is an over-approximation of the set of states reachable in i steps ($i < k$). It is guaranteed that the invariant is true when a fixed-point is reached. The motivation of using interpolation instead of k-induction is that the depth k needed for the first one is, in general, significantly smaller. However, not all bounded model checkers implement the interpolation algorithm.

We propose to strengthen the state transition relation of the system such that, for a sufficiently large k, system executions starting from unreachable states (spurious counterexamples) are ignored during the induction step. Note that, although our method resembles invariant strengthening, it does not bound the state space (by ruling out unreachable states) but limit the space of trajectories instead. Our technique is bottom-up in the sense that it does not depend on the property under verification. The previous approaches (except of interpolation) do not contradict with strengthened transitions, hence, hybrid algorithms can be directly derived. For example, properties proven via strengthened state transitions can be used as lemmas in subsequent proofs, or disjunctive invariants can be proven based on a system with a strengthened transition relation. Interpolation, on the other hand, uses fixed-point calculation rather than induction as proof method; therefore, techniques towards inductive invariants are not directly applicable here. Note that our method, if successful, runs k-induction only once, whereas other induction-based approaches use failed proofs or pre-analysis before the property can be proven.

We present a general framework which uses k-induction as verification method and strengthens the state transition relation of the system. We prove that the framework, when used for verification, (i) might enhance the completeness of k-induction and (ii) preserves soundness. Analogous to invariant strengthening, the main challenge is to invent how to obtain the strengthened state transition relation. As a special case, we show how the Boolean guards of the system's transitions can be modified to strengthen the state transition relation. For this method to work, we make the assumption that the control flow of the system is (at least partially) known. This technique is demonstrated through the next simple example.

1.2 A Short Example: the Bakery Protocol

We show how strengthened guards can be used to model check a simplified version of Lamport's Bakery protocol [11]. We use SAL's implementation of k-induction [8] which applies none of the techniques mentioned above. As the Bakery algorithm works with infinite domains (the ticket number of a process can be any natural number), we use SAL's infinite bounded model checker (sal-inf-bmc) for verification[2]. The

[1]The parameter l can be tuned such that the BDD computation is still effective.

[2]sal-inf-bmc makes calls to the Yices [8] Satisfiability Modulo Theories (SMT) solver which can also handle infinite

```
1 bakery: CONTEXT = BEGIN
2  PC: TYPE = {sleeping, trying, critical};
3  job: MODULE =
4    BEGIN
5      INPUT y2 : NATURAL
6      OUTPUT y1 : NATURAL
7      LOCAL pc : PC
8      INITIALIZATION
9                  pc = sleeping;
10                 y1 = 0
11    TRANSITION
12      [ pc = sleeping --> y1' = y2 + 1;
13                          pc' = trying
14        []
15        pc = trying AND (y2 = 0 OR y1 < y2)
16 AND y1 > 0          %STRENGTHENED GUARD
17            --> pc' = critical
18        []
19          pc = critical --> y1' = 0;
20                          pc' = sleeping ]
21  END;

22  system: MODULE =
23   job
24   []
25   RENAME y2 TO y1, y1 TO y2 IN job

26  mutex:THEOREM system
27   |- G(NOT(pc.1 = critical AND pc.2 = critical));
28 END
```

Figure 2: The SAL model of the Bakery protocol

SAL code of the protocol with two processes is depicted in Figure 2. The protocol's main property, mutual exclusion (`mutex`), is defined as an invariant saying that it is never true that both processes are in the critical section at the same time.

SAL cannot prove `mutex` using the default depth 10. A spurious counterexample of length 10 is produced starting from a state where the 1^{st} process has `pc=trying`, `y1=0`. This is not a reachable state because every process increments its ticket when changing from `sleeping` to `trying` (line 12-13). As a result, the system reaches a state which violates mutual exclusion. Note that the original model cannot be proven with any depth. Therefore, it does not help increasing k to any large number. This can be seen by considering that the state where the two processes have respectively `pc.1=trying`, `y1=0` and `pc.2=trying`, `y2=1` is a recurring one because the 2^{nd} process returns to the same state if it first enters the critical section. As a result, a counterexample similar to the previous one can be produced for any value of k. We propose to strengthen the state transition relation such that the spurious counterexample is not a possible run of the system. We do it by simply adding a new clause to the guard of the transition that drives the process into the critical section (line 16). The new condition requires that `y1` is always positive when the process is at `pc=trying`. The theorem can now be proven.

Our observation is that the previous solution gives rise to a general approach if the system follows a strict control flow. The idea is to use the updates in the previously executed transitions for strengthening the guards. In particular, each Bakery process periodically alternates between `sleeping`, `trying` and `critical`. Therefore, we can safely state (without changing the specified behavior) that the assignment of `y1` at line 12 is still valid at line 16.

Paper structure. First we establish a general framework of transition strengthening (Section 2), then we present strengthened guards as a special case (Section 3). Finally, we use strengthened guards to formally verify a diagnosis protocol with more than 150 lines of SAL code and compare our technique with other approaches (Section 4).

2. A SIMPLE FRAMEWORK

We first formally define the system, its properties and the applied proof method (Section 2.1), then a general technique is presented and its properties are formally proven (Section 2.2).

2.1 Preliminaries

Assume that the *system* is defined as a general state transition graph with tuple $M = (S, I, R, L)$. As usual, S denotes the set of states, $I \subseteq S$ the set of initial states, $R \subseteq S \times S$ the state transition relation and $L : S \rightarrow 2^{AP}$ the labeling function (AP is the set of atomic propositions). For simplicity, we use the notations $I(s)$ iff $s \in I$, $R(s, s')$ iff $(s, s') \in R$. A *path* in M is a sequence of states $s_0, ..., s_k$ iff $R(s_i, s_{i+1})$ for all $0 \leq i < k$. Note that this definition of path does not require the first state to be an initial state. We use the predicate $path(s_0, ..., s_k)$ to designate paths. Furthermore, we define the set of *reachable states* in M as $Reach_M = \{s | I(s) \vee \exists s_0, s_1, ... : I(s_0) \wedge path(s_0, ..., s)\}$.

Properties are formulas that are defined based on the formal definition of the system. We restrict to *state invariants* (or simply invariants) which are true in all reachable states. Formally, invariants can be defined over AP using the standard Boolean operators. We use the shorthand $P(s)$ such that the predicate is true iff p holds in s, i.e., the atomic propositions in $L(s)$ satisfy p. A state s is called *P-state* iff $P(s)$ holds.

We assume that k-induction is used to verify invariant p in system M. Before giving the formal definition of k-induction, we describe it informally. The intuition is to check that paths of length k starting from an initial state visit only P-states. If it is not always the case, a counterexample of length k is found. The dilemma is to determine the value of k such that the invariance of the property can be safely established. The strategy is to check whether there exists a $(k+1)$-long path starting from an arbitrary state and leading to a non-P-state. If not, we can safely stop since all shorter paths have already been checked. Otherwise, the value of k needs to be increased [17]. Note that this method corresponds to the generalization of the simple induction rule which first states that the invariant includes all initial states (base case), then it proves that the invariant is closed on the transitions (inductive step). For the formal definition of k-induction we use the one taken from [9]. The proof is parameterized with a system M, depth k and property p. A

domains. We note that in the context of this paper, SAT and SMT solvers are conceptually identical in that they both decide satisfiability of logical formulas.

k-induction proof instance is denoted by $IND_M(k)(p)$ and is true if the following two predicates hold for all $s_0, ..., s_k$.

Base case: $I(s_0) \wedge path(s_0, ..., s_{k-1}) \rightarrow P(s_0) \wedge ... \wedge P(s_{k-1})$

Induction: $P(s_0) \wedge ... \wedge P(s_{k-1}) \wedge path(s_0, ..., s_k) \rightarrow P(s_k)$

Note that there is no need to explicitly search for paths shorter than k if we can assume that the system is live. In this case, violating paths of length $< k$ are also checked by $IND_M(k)$.

Finally, a proof method is called *sound*, if the fact that it proves p in M implies that p is indeed true in M. It has been shown that k-induction is sound with respect to state invariants [17, 9]. Furthermore, a proof method is *complete* if every property p which is true in M can be proven by the method. There have been several attempts towards the completeness of k-induction. In worst case, exhaustive exploration can be used if there is an upper bound on the length of the possible trajectories of the system. It is often possible to find such an upper bound even in systems with infinite paths. For example, path compression can be used to make sure that a trajectory only visits "new" states [9]. As the most common special case, only loop-free paths are considered. In this case, an alternative induction step is to check that no $(k+1)$-long loop-free path starts form an initial state [17]. Consequently, we can conclude that all loop-free paths are covered by the base case and the entire reachable state has been explored. Formally, the definition of *path* predicate needs to be modified if compressed paths are used. Although our technique works with all known optimizations, it is not dependent of any of them, therefore, they are omitted in the formal discussion.

2.2 k-Induction with Strengthened Transitions

We define a new system based on the original specification which will be used during the verification. The intuition is to constrain the state transition relation such that the observable behavior of the system remains the same. In this paper, we concentrate on state invariants, therefore, the preservation of the behavior corresponds to having the same set of reachable states in both systems. Next, we define systems with strengthened transitions in a declarative way, i.e., without discussing how the conditions can actually be fulfilled. We will show an implementation of such systems in Section 3.

Definition 1. $M' = (S, I, R', L)$ is a *system with strengthened transitions* with respect to $M = (S, I, R, L)$ if $R' \subseteq R$ and $Reach_M = Reach_{M'}$.

The next simple theorem claims that a system with strengthened transitions might improve but never weakens completeness of k-induction. This is equivalent with showing that the set of provable invariants in M is never greater than that in M'. In general, full completeness is not reached, therefore, both sets are a subset of the set of all invariants in M.

THEOREM 1. *The completeness of k-induction in a system with strengthened transitions can be characterized as follows.*

$$\{p|IND_M(k)(p)\} \subseteq \{p|IND_{M'}(k)(p)\} \subseteq \{p|M \models \mathsf{G}p\}$$

PROOF. Assume that p is an invariant. From Definition 1, M and M' entail the same set of reachable states. Therefore, the base case of $IND_M(k)(p)$ is true iff it is true in $IND_{M'}(k)(p)$. As $R' \subseteq R$, $path(s_0, ..., s_k)$ in M' implies $path(s_0, ..., s_k)$ in M, thus, $IND_M(k)(p)$ implies $IND_{M'}(k)(p)$. We have proven that if k-induction can verify p in M it can also prove it in M'. We now show that it is possible that $\{p|IND_M(k)(p)\} \subset \{p|IND_{M'}(k)(p)\}$. Assume that $\neg P(s_k)$, $path(s_0, ..., s_k)$ in M for some $s_0, ..., s_k$ but $\neg path(s_0, ..., s_k)$ in M' for any $s_0, ..., s_k$. This is possible if $R' \subset R$. Finally, we prove that general completeness cannot be guaranteed. Assume $\neg P(s_k)$ and $path(s_0, ..., s_k)$ in M'. In this case, $IND_{M'}(k)(p)$ is false even if p is an invariant. □

2.3 Discussion

Refutation. In general, better completeness with strengthened transitions comes at a price. Since the new transition relation is restricted with respect to the original one, it is possible that the shortest path between two reachable states s_0 and s increases from k (in M) to $k' > k$ (in M'). As a result, the base case of the induction might find counterexamples with greater depths. For example, assume that $P(s)$ does not hold and k' is the length of the shortest path to s in M'. In this case, a counterexample in M' can only be found with depth k', whereas, it suffices to use depth k in M. As finding counterexamples quickly is particularly important in early phases of the system development, we propose using M for refutation and M' for verification. However, making this differentiation is not always needed. In the next Section, we show an implementation of transition strengthening which never degrades the method's ability to disprove invariants.

Soundness. We remark that our method, while improving completeness of k-induction under favorable circumstances, preserves the general soundness property. This directly comes from (i) the restriction to state invariants and (ii) the condition that the new system must entail the same space of reachable states. Accordingly, if a state s is reachable in M it is also reachable in M', therefore, any state invariant is true in M iff it is true in M'. The last statement and the fact that k-induction is sound imply that the verification of M via M' is sound.

3. AN IMPLEMENTATION

Previously, we introduced a theoretical framework to enhance the completeness of induction-based BMC. The main premise was that it is possible to create a system with strengthened transitions. But how can we do it in practice? In this section, we give a solution as a possible implementation of the general framework. The heart of the solution is the assumption that a fixed control flow of the system exists.

3.1 Preliminaries

To ease further discussion, we use interpreted first order formulas to describe the system. The predicates and function symbols as well as the translation from (and to) the tuple-based representation of Section 2 are defined similarly to those in [6]. Let $V = \{v_1, ..., v_n\}$ be the set of *system variables*. Variables range over a finite *domain* D. A state $s : V \rightarrow D$ is interpreted as a valuation of V assigning values from D to a subset of variables. $\mathcal{I}$ denotes a first order formula which is only true for valuations representing *initial states*. $\mathcal{R}(V, V')$ is the formula which corresponds to the *transition relation*. Variables in V are thought of as current state variables and those in V' are the next state variables. The formula is true if the valuations of V and V' represent a current and next state of the system. Finally, *properties* are defined based on atomic propositions of the form $v = d$ where $v \in V$ and $d \in D$. A proposition is true in a state s if $s(v) = d$.

Using *guarded commands* is a common way to describe a system's transitions. The simple example of Figure 2 also used guards to rule the execution of transitions. The next values of variables can be updated whenever a guard condition based on current values is true. In general, we can assume that the transition formula implements the following template: $(guard_1 \wedge update_1) \vee (guard_2 \wedge update_2) \vee ...$, where $guard_i$ is restricted to be defined over V. We say that the system is defined with guarded commands if $\mathcal{R}(V, V')$ implements the previous template. In addition to the definition of all possible state transitions, a *scheduler* is attached to the system to decide which transitions are executed and in which order. For example, it is the scheduler's job to resolve conflicts when more than one guard is enabled. The SAL scheduler, for example, selects one of the enabled transitions non-deterministically or, if none of the guards is true, it executes the default transition if it is defined (using the `ELSE` keyword). The system deadlocks if no transition can be executed.

3.2 Execution Plan: Fixed Control Flow

Our implementation of systems with strengthened transitions is based on the assumption that the control flow of the system is a-priori known. For example, in the Bakery protocol each process executes the same transitions in the same order. More generally, we assume that an *execution plan* is available which contains information about the order of executed transitions. Let $trans_i$ denote the formula $(guard_i \wedge update_i)$. In the simplest case, an execution plan is a sequence of positive integers $i_1, i_2, ...$ meaning that $trans_{i_j}$ is the j^{th} transition executed by the system. We call it *total* execution plan. However, even if a system adheres to a control flow, partial non-determinism is possible. Therefore, we define *partial* execution plans to be a similar sequence of numbers with the guarantee that $trans_{i_k}$ is executed after $trans_{i_j}$ iff $j < k$. We allow multiple partial execution plans for the same system, however, with the restriction that they contain distinct transitions. This makes sense otherwise there was an uncertainty about what transitions are preceded by the one appearing in multiple plans. For example, the Bakery protocol with two processes can be associated with two partial execution plans each of them containing the three transitions between lines 12 and 20 (Figure 2). Note that it is not possible to associate a total execution plan with the Bakery protocol as the participating processes execute transitions asynchronously without an a-priori global order of transitions.

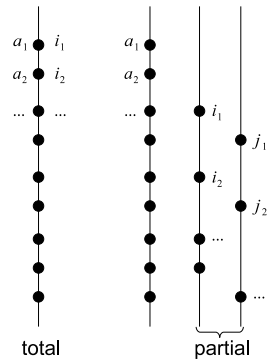

Figure 3: Execution plans and actual executions

Denote the sequence $a_1, a_2, ...$ an *actual execution* where $trans_{a_i}$ is the i^{th} transition executed by the system. The actual execution models an arbitrary run of the real system. The relation between execution plans and actual execution is depicted in Figure 3.

3.3 Strengthened Guards

The main idea of strengthened guards is to prune some of the impossible behavior by taking into account the scheduling of the transitions in the real execution of the system. We only make a natural assumption to require that a transition can only be executed if its guard is true. To eliminate impossible runs we use the updates of previously executed transitions to strengthen the guard. If the system adheres to the execution plan the new guard will be true exactly when the old one is true because no further updates have been done to the variables. In k-induction, the values of state variables in spurious counterexamples are usually not in accordance with the execution plan. Consequently, such runs can be ruled out by using strengthened transitions.

The first solution is to replace the original guard $guard_i$ with $guard_i \wedge (\bigvee update_{i_{j-1}})$ for all $i_j = i$ in the execution plan. It is possible that a transition is preceded by different transitions in the execution plan. Therefore, we add all of them such that only one needs to be true. If a transition is executed as first, the possible initial assignments can also be added to the strengthened guard. Since execution plans are disjunct or total, at most one execution plan is used for the replacement of a guard. Before formally defining the strengthened system, consider the following two issues.

- Formally, $update_i$ is defined over V and V'. Therefore, its syntactical rewriting is needed when used in a guard because guards can only use current state variables. For example, the update `y1'=y2+1` can be used as `y1=y2+1` in the strengthened guard. However, this is not correct if current state variables that are used in the assignment are modified by the same update. In the previous example, changing the value of `y2` would

mean that `y1=y2+1` will not be true. In general, if variable v is used as both current and next state variable by the same update, its old value stored in an (auxiliary) variable *v_old* can be used in the strengthened guards (e.g., `y1=y2_old+1` in the previous case).

- We have to be careful with partial execution plans because transitions that are not part of the plan can modify the assignments with respect to the ones in $update_{i_{j-1}}$. For example, the update `y1=y2+1` could not be directly used to strengthen the guard in line 15 (Figure 2) because the other process might change `y2` in the meanwhile. Therefore, for the general case, we assume that there is a function fun which extracts the possibly strongest condition which can be safely used to strengthen the guard; fun takes an update as input and returns a formula over V. For example, in the strengthened version of the Bakery protocol, $fun(y1' = y2 + 1)$ returns $y1 > 0$ to use it for strengthening the guard at line 15. Techniques for implementing fun in an effective and automated way are part of our future work (see Section 3.6 for more).

The system with strengthened guards M' can now be defined based on the original specification M of the system. Note that the transformation from M to M' can be automated if we suppose that the function fun and the execution plans are given.

Definition 2. Supposed that executions plan(s) $i_1, i_2, ...$ of a system with guarded commands, described with $\mathcal{I}$ and $\mathcal{R}(V, V')$, are available; the system with *strengthened guards* is described with $\mathcal{I}$ and $\mathcal{R}'(V, V')$ such that every $guard_i$ in $\mathcal{R}(V, V')$ is replaced by $guard_i \wedge (\mathcal{I} \vee \bigvee fun(update_{i_{j-1}}))$ for all $i_j = i, j > 1$. The strengthened guard only includes $\mathcal{I}$ if $i_1 = i$.

We claim that Definition 2 is a special case of Definition 1. Therefore, the result of Theorem 1 and the discussion in Section 2.3 is valid.

COROLLARY 1. *Assume that a system A is defined with guarded commands. The system A with strengthened guards (call it A') is a system with strengthened transitions with respect to A if A' is scheduled as A.*

PROOF SKETCH. Assume that A and A' are represented by tuples $M = (S, I, R, L)$ and $M' = (S, I, R', L)$ respectively. This assumption is valid because Definition 2 only modifies $\mathcal{R}(V, V')$. By intuition, $R' \subseteq R$ because guards are never weakened and A and A' share the same schedule (see Section 3.4 for more details). We have to prove that $Reach_M = Reach_{M'}$. Assume that the execution plan $i_1, i_2, ...$ is total. In this case, $fun(update_{i_j}) = update_{i_j}$. For every reachable state s in M, there is $s_0, s_1, ...$ such that $I(s_0)$ and $path(s_0, ..., s)$. We use induction by the position of states in the path. $I(s_0)$ is true in both M and M'. s_j is computed by executing $guard_{i_j} \wedge update_{i_j}$. The same s_j can be computed by $trans_{i_j}$ in M' if $guard_{i_j} \wedge update_{i_{j-1}}$ (the strengthened guard) is true. The condition is indeed true because $update_{i_{j-1}}$ represents the assignments in s_{j-1} which is the current state. If the current state is an initial state, $guard_{i_1} \wedge \mathcal{I}$ is true. If the execution plan is not total, the proof is dependent of the implementation of fun. Assuming that $fun(update_{i_j}) = update_{i_j}$, the proof is similar to the one presented above. □

Note that our technique does not eliminate transitions, it only strengthens the guards. Since every new guard is trivially true on paths starting from initial states (this is guaranteed by construction), valid *counterexamples* can be found with the same depth as for the original system. Therefore, unlike in the general case (see Section 2.3), there is no need to use the original system for seeking counterexamples. In the next Section, we explain why Corollary 1 makes the assumption on the schedule of transitions.

3.4 Preserving Execution Semantics

We say that a system with strengthened transitions A' is scheduled as the original system A if they execute the same sequence of transitions if possible. This ensures that if a strengthened guard is not true then no other transition is taken and the execution stops, i.e., no path of depth k exists.

Note that the scheduler which triggers the execution of transitions might prevent the previous condition from being true. It is the core of our technique that the execution stops if a transition which is supposed to be executed is not enabled. However, the scheduler might select another transition which can be alternatively executed. In SAL, for example, having the default transition might cause the strengthened system to explore states that are not entailed by the original system. There are two options to circumvent the undesired interplay of the scheduler. In theory, it is possible to directly translate the specification into the tuple-based representation without using the built-in scheduler of the execution environment. This solution is cumbersome and can only be viable if the process of translation is fully automated. Another option is to use the scheduler at hand and enforce it to implement the desired schedule. Usually it is not a hard thing to do. For example, ELSE-branches can easily be eliminated from the original SAL specification by replacing them with regular guarded commands. We show an example of how to do it in Section 4.

3.5 Optimizations

Usually, an update only changes the assignment of a real subset of all state variables instead of changing each of them. Therefore, guards can be further strengthened with updates that are not defined by direct predecessors of a transition plan but whose assignments (or some of them) are not changed by the subsequent transitions.

Another refinement of our basic technique is to strengthened guards based on variables (say *v_old*) that store old (e.g. the previous) values of another variable (v). In this case, the assignments of v can be used in guards even if value of v has already been re-assigned. It is also possible to store more than one old value (i.e., from different states) of the same variable. In this case, a good tradeoff has to be found between the overhead of the verification and the benefit of using strengthened guards. Note that *v_old* is not necessarily

an auxiliary variable of the system that is only introduced to implement strengthened guards. It is a common technique to use such variables in order to express liveness properties as invariants. Variables storing earlier values of other variables can also be a functional part of the system's model if some correlation between the current and previous values needs to be established. Our case study shows an example of both optimization techniques.

3.6 Discussion

Transition-validated assertions. The technique of transition validated assertions (TVAs) is a bottom-up approach for strengthened invariants [13]. By definition, a TVA is an invariant q such that $update_i \rightarrow q$ for every $trans_i$ interfering with q. Accordingly, every TVA is 1-inductive and can be used to strengthen the original property p. Our technique of strengthening the guards based on a fixed control flow can be expressed via TVAs as well. Assume that the transitions of the system are uniquely labeled. For example, $trans_i$ can be labeled by i. In the Bakery example, the values of `pc.1` and `pc.2` provide an appropriate labeling. Labels serve, besides providing the means of identifying statements, as locations of control. Note that a location can designate more than one label at the same time. For example, if the control of the Bakery protocol is at `pc.1=sleeping`, it might be also at `pc.2=trying`. Let the auxiliary predicate $l(i)$ be true iff the control is at $trans_i$. For simplicity, denote the strengthened guard of $trans_i$ by $guard_i \wedge (\bigvee rewr\_update)$. Accordingly, the formula $l(i) \rightarrow (\bigvee rewr\_update)$ is a TVA because every update that might lead to location described by $l(i)$ is included in $(\bigvee rewr\_update)$. This is guaranteed by the execution plan. It can be seen the formula is still 1-inductive if our optimization of using earlier updates to strengthen a guard is applied (see Section 3.5).

Our work differs from TVAs in that it is presented in the context of strengthened transitions. Even though strengthened guards and TVAs are logically equivalent, their implementation can entail different overheads as shown in our case study. Furthermore, we use execution plans to derive strengthened guards which can be automated or easily implemented by the user.

Control flow-determinism. One might argue that an execution plan about the system does not exist or is unknown. In fact, strengthened guards are useless in the former case. However, we believe that a static sequence of actions in the system's execution can be observed for a variety of applications. For example, safety-critical embedded systems utilize deterministic protocols like diagnosis, membership [16] or startup [19] for predictability. We also think that execution plans can be, in many practical cases, easily determined based on the system's specification. For example, the execution plans of the Bakery protocol can be obtained from its original description [11] without the need of understanding how the protocol works: the specification of each process is given as a sequence of actions which corresponds to partial execution plans. In other cases, the implementation restricts the unconstrained schedule of the high-level formal description. For example, the compiler injects control flow information of the hardware design with respect to the abstract functional specification on which the formal analysis is performed [18]. Therefore, the guards in the specification can be strengthened if we know what the compiler does in synthesizing the schedule. The automation of determining the control flow is part of our future work.

Update rewriting. Although the implementation of fun is kept open as future work, we speculate about some aspects of its possible automation. Given an update upd, we look for the result of $fun(upd)$ to strengthen the guards in accordance with the execution plan(s). Call a variable in V' *stable* with respect to upd if its value remains unchanged until the execution of the strengthened transition. Otherwise, the variable is called *unstable*. Now, $fun(upd)$ is the identity function if all variables of V' appearing in upd are stable. This means that upd can be used in the strengthened guard after the trivial syntactical rewriting described in Section 3.3. Otherwise, the empty constraint ($true$) can be used which corresponds to discarding upd. Decision procedures can be used to obtain more sophisticated solutions. For example, unstable variables in upd can be replaced by symbolic constants and automatic static analysis can be used to derive a provably valid but non-empty constraint. In the example of the Bakery protocol, considering the domain of the variables, $fun(y1' = y2 + 1) = (y1 > 0)$ can be computed without knowing the value of $y2$.

4. EXECUTION PLANS IN SAL: VERIFYING A REAL PROTOCOL

As a proof of concept, we use execution plans to verify a diagnosis protocol with the SAL model checker. The techniques of strengthened guards, transition-validated assertions and lemmas are compared. Our experiments show that all techniques are able to prove the properties that are non-inductive in the original model. However, additional overhead is induced due to the manipulation of the model. Using strengthened guards, the pure execution time of the k-induction rule is the shortest among all applied techniques.

In the remainder of this Section, we first briefly describe the subjected protocol and its implementation in the SAL language (Section 4.1), then we present how different techniques can be implemented to reduce the depth of k-induction based on the a-priori known control flow of the system (Section 4.2). Finally, we compare the performance of these techniques using the BMC model checker of the SAL environment (Section 4.3).

4.1 Diagnosis with Hybrid Faults

Diagnosis is a service that is able to locate faults in a system. In our case study, we use a distributed diagnostic protocol applicable in *synchronous* environments[3]. The execution model is that the nodes proceed through a parallel sequence of rounds such that each round is split into communication and computation phase. The algorithm is based on round-based consensus with malicious (aka. Byzantine) faults [12]. The plain model of worst-case faults is augmented with other less severe faults such as *benign* faults or *symmetric* value

[3]Communication based on message sending is assumed where correct nodes are able to send and deliver a message on time.

faults. A benign node executes the protocol but it is unable to send a message or sends wrongly formatted data. A symmetric node is malicious faulty with the restriction that it sends the same message to everyone. The benefit of the hybrid fault model is that it entails enhanced fault tolerance with the same number of replicas. The protocol used in this case study, called hybrid diagnosis (HD), executes consensus on the local syndromes about the health status of the system [20]. It guarantees that, under the fault hypothesis, (i) all non-faulty nodes agree on the diagnosis of the system (*consistency*), (ii) all benign faults are eventually detected (*completeness*) and (iii) non-faulty nodes are never diagnosed faulty (*correctness*). The fault hypothesis defines that the overall number of nodes N is greater than $(2a + 2s + b + 1)$ where a, s and b denote the number of malicious, symmetric and benign faults, respectively.

The basic assumption of the protocol is, besides synchrony, that distributed nodes can run computation and send/receive messages in parallel with each other. This might not be the case in systems where resources are shared to reduce cost. Consequently, the protocol (e.g., [16]) and its formal model must be modified to accommodate the conditions. We remark that the latter is not always needed. For example, the same model of the presented protocol can be used even if the system uses a shared communication bus and if different applications run on the same computer [4]. This is achieved by using an abstraction and showing a bi-simulation between the abstracted and the original models.

The protocol. The SAL language allows the user to define *modules* which can be thought of as building-blocks of the overall model. The model of the protocol contains two modules, one describing the fault model, the second defining the protocol's operations executed by the distributed nodes. As every node is supposed to execute the same code, the module defining the protocol is parameterized with the ID of the corresponding node. We use SAL's guarded commands to identify the different stages of the protocol and to execute the part of the code which is associated with that stage. Additional stages are defined when auxiliary operations (e.g., fault generation) are being computed which cannot overlap with the protocol's operation. In accordance with the synchrony assumption, the modules are composed together using SAL's *synchronous composition*. This ensures that transitions are executed in a lock-step manner, i.e., the execution stops unless an enabled transition can be selected in every module and the selected transitions can be executed following parallel execution semantics (non-conflicting updates, etc.) [7]. Recall that the Bakery example of Section 1 used asynchronous composition, where only one transition is selected and executed at each step of the model[4].

The protocol HD is a sequential and periodic execution of the following two rounds:

[4]We remark that SAL's different composition semantics can be considered as instructions for compiler about how the transition relation is calculated.

Variable	Stage	Dependency	Valid at stage
`ls_symm'`	0	`fvec`	1-3 (+0 w/ `fvec_old`)
`ls'[i]`	0	`fvec`	1-3 (+0 w/ `fvec_old`)
`sm'[i]`	1	`fvec` `ls[1..N]` `ls_prev[1..N]`	2-3 (+0 w/ `fvec_old`)
`chv'[i]`	2	`sm[i]`	3,0,1
`fvec'`	3	`fvec`	0,1-3 (all w/ `fvec_old`)

Table 1: Variables and their dependencies in the SAL model of the HD protocol: N is the number of nodes, `var[i]` denotes `var` at node i and the primed version of a variable means its next value.

Round 1

1. *Sending workload (communication):* every node i broadcasts a message.
2. *Local detection (computation):* every node i diagnoses the other nodes based on the messages received from them and forms a *local syndrome* (`ls[i]`) indicating the health status of each node. The j^{th} value of this vector is 0 if node j is diagnosed faulty and 1 if it is correct.

Round 2

2. *Global dispersion (communication):* every node i broadcasts its local syndrome.
3. *Global assimilation...:* the local syndromes received from the other nodes, together with the syndrome obtained in Round 1, are compiled into a *syndrome matrix* (`sm[i]`). The j^{th} row of the matrix is the local syndrome received from node j.
4. *...and analysis (computation):* Every node i computes hybrid majority on the values of each column in the syndrome matrix and derives a *consistent health vector* (`chv[i]`).

The protocol uses a special form of majority function, called *hybrid majority*, where wrongly formatted (i.e., semantically incorrect) local syndromes and the node's opinion about itself are omitted in the voting; the default outcome is "correct" if no value is in majority.

Table 1 depicts the variables that are used to encode HD in the SAL language[5]. The different stages indicate the protocol's steps and auxiliary operations. The current stage is stored in a control variable called `pc` (program counter); this variable is re-set to 0 when it reaches 3 to enable periodic execution. In addition, the following variables are defined: the fault vector (`fvec`), its old value (`fvec_old`)

[5]The source of models in this Section are available at http://www.deeds.informatik.tu-darmstadt.de/peter/sal/sources. The model of the HD protocol can be found under diagnosis.sal.

and a variable to store semantically incorrect but symmetrically disseminated local syndromes (`ls_symm`). The fault vector is updated at the end of each round such that the fault hypothesis is satisfied. It is assumed that, in each instance of the protocol, a node can be faulty according to at most one fault class. For example, it cannot be that a node is benign faulty at round 1 and Byzantine during round 2. The variable `ls_symm` contains the local syndrome that a symmetric faulty node sends to the other nodes. In our experiments, we only verify systems with $N < 6$; therefore, it suffices to have just one such a vector since s (the number of symmetric faults) is at most one. Note that overlapping instances of the protocol can be executed. This means that in every communication round k instance A executes Round 2 and another instance B launches the protocol by executing Round 1. Consequently, in our SAL model, the syndrome matrix of node i (`sm[i]`) is not updated based on the local syndromes of this round but on those from the last round (`ls_prev[1],...,ls_prev[N]`).

Properties. We define all three properties of the HD protocol: consistency, correctness and completeness. Consistency can be naturally defined as a state invariant because it requires that the consistent health vector is consistent among the nodes at the termination of each instance, i.e., at stage 3. Since the protocol terminates after Round 2, liveness properties have to be verified with one round delay. Accordingly, correctness requires that no node that was correct during round $(k-1)$ is diagnosed as faulty in round k; completeness means that every benign fault occurring in round $(k-1)$ must be detected at round k. It is possible to define correctness and completeness as invariants as well, if we store the fault vector of the previous round. For example, diagnosis can be defined as follows (`chm[i]` corresponds to `chv[i]`):

```
diagnosis_completeness: THEOREM system |-
    G(FORALL(l,m:nodes):pc=3 => (
        fvec_old[l]=benign => chm[m][l]=0));
```

4.2 Inductive Invariants with Execution Plan

If the control flow of the system is a-priori known, the system's model can be modified such that the properties of the system (expressed as invariants) are more likely to inductive. The presented model of the HD protocol gives rise to a total execution plan. This is because the protocol is fully deterministic, the modules of the SAL model are connected via synchronous composition and the auxiliary operations (the updates of `fvec` and `ls_symm`) do not cause non-determinism in the control flow. Given that one is implemented via four stages, the j^{th} transition in the execution plan is always the transition guarded by stage j modulo 4. Note that, although the existence of total execution plans requires that the control flow is deterministic, the data flow can contain non-determinism. For example, our model non-deterministically generates local syndromes of Byzantine nodes, the activation of faults, etc.

Strengthened guards. We have implemented a model of the HD protocol with strengthened guards. The complete model is available in the source file diagnosisSG.sal. Now, we discuss the outline of our implementation. The technique of strengthened transitions is based on restricting the global transition relation. In case of synchronously composed modules, the global transition relation can be derived from the corresponding transitions of the modules. Since a module is only allowed to update local variables (which might depend on input variables), it is possible to strengthen the guards "locally" in each module. For example, this is how we strengthen the guard `pc=3` with the assignment of `chv` from the last transition (note that the update part of the transition is empty because stage 3 is an auxiliary transition to update the fault vector in the fault module):

```
pc=3
%---- Guard strengthening STARTS --
AND chv=[[n:nodes] h_maj(sm)[n]]
%---- Guard strengthening ENDS ----
-->
```

We do not use SAL's ELSE branch in the model to preserve execution semantics. The program is forced to execute the transition (at stage 0,1,2 or 3) determined by the execution plan. Otherwise, the program's execution is blocked. Once again, this does not happen in regular runs, i.e., when the the program is started from a proper initial state. We applied the optimization when updates that are not in direct predecessors of a transition are also used to strengthen the guard of that transition. Table 1 helps determining the strengthened guards that can be used at different stages of the protocol. The third column depicts the dependencies of each state variable. An update of a variable can be used to strengthen a guard until none of its dependencies is re-updated. For example, `chv` is dependent of `sm` which is updated at stage 1. Consequently, the update of `chv` at stage 2 can be used from stage 3 to stage 1 in the next round. Furthermore, the updates of variables depending on `fvec` can be used in guards even after `fvec` is updated because its old version (`fvec_old`) is also available. Finally, we note that this example of guard strengthening assumes the trivial implementation of fun where $fun(upd) = upd$. This is because the execution plan is total and the optimization takes the dependencies into account.

Transition-validated assertions. We have strengthened the properties of the HD protocol with transition-validated assertions that can be obtained through the total execution plan. The model and its properties are available in the source file diagnosisTVA.sal. For a fair comparison, we used the same assertions that appear in the strengthened guards of the previous model. For example, the following assertion corresponds to the guards strengthened at stages 3, 0 and 1 with the update of `chv`.

```
pc/=2 => FORALL(m:nodes):
    chm[m]=[[n:nodes] h_maj(sm_vec[m])[n]]
```

Note that auxiliary variables are used to refer to local variables of parameterized modules (e.g., `sm_vec[i]` corresponds to `sm[i]`). The same model can be used for invariance checking with lemmas. We defined a lemma called `TVAs` which is comprised of the TVAs used to strengthen the properties.

Technique	Depth	EFA (s)	UQ (s)	BPCT (s)	BFMC (s)	BIF (s)	SAT time (s) (*)	Overall time (s)
-	7	0,11	0,07	0,01	0,23	0,48	0,38	1,46
SG	3	0,24	0,22	0,04	0,42	0,65	0,26	2,52
TVA	3	0,14	0,2	0,3	0,2	1,77	26,22	29,18
Lemma	3	0,15	0,21	0,3	0,2	0,83	9,44 (†)	11,17 + 1,59 (‡)

Table 2: Results of proving consistency of the HD protocol in a 4-node system by using sal-bmc (with Yices SAT solver). Abbreviations: Expanding function application (EFA), unfolding quantifiers (UQ), Boolean property conversion time (BPCT), Boolean flat module conversion (BFMC), building induction formula (BIF) as the time to build the base formula plus the time to build the induction formula. (*) It is the sum of Yices execution time for the base case and the inductive step. (†) The inductive step took 97% of SAT time. (‡) The lemma `TVAs` was proven in 11,17 seconds at depth 1; the consistency property was proven in 1,59 seconds at depth 3. All data in this row correspond to the proof of the lemma.

4.3 Experiments

Reduced induction depth. We verified the properties of the HD protocol in systems where $N < 6$ by using the models presented in the previous Section. For example, consistency can be proven at depth 3 in the model with strengthened guards. The original model fails to prove the property with the same depth and returns a spurious counterexample. In our setting, the counterexample starts in a state at stage 0 where the fault hypothesis is violated by fault vector. In addition, the local syndromes do not correspond to the fault status of the system. Consequently, since the syndrome matrix and the consistent health vector are computed based on data that cannot appear in the assumed system, consistency is violated. In the model with strengthened guards, the transition at stage 0 is strengthened with the assignments of the fault vector and the local syndromes. This entails that runs similar to the previous one are ruled out in the inductive step of 3-induction and the property can be proven. We remark that strengthened guards might be useful even if the property was not inductive. In fact, as a result of guard strengthening the counterexample resembles more a valid run and appear less "chaotic" than in case of the original model and it helps the user to better understand what goes wrong in the run. We expect that the other techniques are able to prove consistency at the same depth. Indeed, consistency strengthened with the established TVAs as well as the original consistency property using our lemma could be verified by 3-induction. In the following, we discuss how the use of these techniques affect the performance of SAL's BMC model checker.

Performance issues. The experiments of verifying consistency, completeness and correctness with different system size showed similar trends. Table 2 depicts the results of proving consistency for $N = 4$. The same induction depths could be measured for completeness and correctness as well. The experiments were run on a single processor of a double-core Intel Xeon 5130 at 2 GHz with 4 GBytes memory. In this case study, the properties of the protocol could be proven even in the original model by increasing the induction depth to at least 7. Table 2 compares the time of verification with the different approaches. As we can see, the original model is still the fastest (in terms of overall time) in spite of the increased depth. However, in general, increasing the depth might not be feasible or might not result in an inductive invariant. We can also see that the SAL implementation of strengthened guards (SG) outperforms that with TVAs. We speculate that this is because of the ability of strengthening the guards "locally", i.e., within a module based. It is very fast to prove consistency by using the lemma; however, the lemma must be proven separately which takes approximately six times more than the proof of the property[6]. The modified models entail more time to expand functions (EFA) and to unfold quantifying operators (UQ) because the strengthened guards and TVAs contain both function application and quantifiers. The TVA and lemma techniques spend more time converting the formulas into a Boolean representation (BPCT) because they use a modified form of the original property. On the other hand, the transformation of the modules into Boolean formulas (BFMC) takes the most time in the SG model because the strengthened guards augment the transition system of the original model. Our approach needs the shortest time among the depth-reduction techniques to build the k-induction formulas (BIF) and to run the SAT solver (SAT time).

5. CONCLUSION

We have presented an alternate technique for making invariants inductive. The proposed framework is general allowing the invention of customized solutions. We have implemented a prototype solution which strengthens the guards of the transitions based on the the assumption the sequence of the executed transitions is known. We have reported the benefit and the overhead of using this method to model check a system that we develop in a parallel project. We remark that the general technique is not restricted to standard induction-based invariant checking [17] but can possibly be used to improve the proof-quality of any method using k-induction. For example, algorithms using fix-point iteration to prove equivalence of circuits have been augmented with k-induction to provide stronger completeness [3]. The resulting algorithm can be further improved by using strengthened transitions. However, as our technique manipulates the transitions, it is limited to induction schemes that work on the unfolding of the state transition relation.

We believe that the usability of strengthened guards heavily depends on the actual system. Therefore, we plan to apply it for the verification of other systems as well. In future work, we would like to elaborate the full automation of strengthened guards. The main issues are to automatically obtain execution plans and to derive strong strengthened guards

[6]The lemma can be proven by 1-induction; in general, TVAs are guaranteed to be provable via simple induction [13].

without user intervention.

6. ACKNOWLEDGMENTS

The authors would like to thank Marco Serafini and Áron Sisak for the valuable comments.

7. REFERENCES

[1] J. Baumgartner, A. Kuehlmann, and J. A. Abraham. Property Checking via Structural Analysis. In *Proceedings of Conference on Computer Aided Verification (CAV)*, pages 151–165, 2002. Springer-Verlag.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC)*, pages 317–320, 1999.

[3] P. Bjesse and K. Claessen. SAT-Based Verification without State Space Traversal. In *Proceedings of Formal Methods in Computer-Aided Design* (FMCAD), pages 372–389, 2000.

[4] P. Bokor, M. Serafini, A. Sisak, A. Pataricza, and N. Suri. Sustaining Property Verification of Synchronous Dependable Protocols over Implementation. In *Proceedings of High Assurance Systems Engineering Symposium* (HASE), pages 169–178, 2007.

[5] G. M. Brown and L. Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *Proceedings of International Conference on Tools and the Construction of Algorithms (TACAS)*, pages 58–72., 2006.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] L. de Moura, S. Owre, and N.Shankar. *The SAL Language Manual*. Technical Report, SRI International, 2003.

[8] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proceedings of Computer-Aided Verification (CAV)*, pages 496–500, 2004.

[9] L. de Moura, H. Rueß, and M. Sorea. Bounded Model Checking and Induction: From Refutation to Verification. In *Proceedings of Computer-Aided Verification conference (CAV)*, pages 14–26, 2003.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of Symposium on Logic in Computer Science*, pages 1–33, 1990.

[11] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communication of ACM*, 17(8):453–455, 1974.

[12] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[14] K. L. McMillan. Interpolation and SAT-based Model Checking. In *Proceedings of Computer-Aided Verification (CAV)*, pages 1–13, 2003.

[15] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In *Proceedings of Computer-Aided Verification (CAV)*, pages 508–520, 2000.

[16] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstaetter, F. Tagliabó, and J. Koch. A Tunable Add-On Diagnostic Protocol for Time Triggered Systems. In *Proceedings of Dependable Systems and Networks (DSN)*, pages 164–174, 2007.

[17] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties using Induction and a SAT-solver. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125, 2000.

[18] G. Singh and S. K. Shukla. Verifying Compiler based Refinement of Bluespec Specifications using the SPIN Model Checker (To Appear). In *Proceedings of Int. SPIN Workshop on Model Checking*, 2008.

[19] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *Proceedings of Conference on Dependable Systems and Networks (DSN)*, pages 189–198, 2004.

[20] C. J. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng.*, 23(11):684–721, 1997.

Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints

Andrea Calvagna
Dip. Ingegneria Informatica e delle Telecomunicazioni
University of Catania - Italy
andrea.calvagna@unict.it

Angelo Gargantini
Dip. Metodi Matematici e Ingegneria dell'Informazione
University of Bergamo - Italy
angelo.gargantini@unibg

ABSTRACT

In this paper we describe an approach to use formal analysis tools in conjunction with traditional testing to improve the efficiency of the test generation process. We have developed a technique for the construction of combinatorial test suites, featuring expressive constraints over the models under test and cross coverage evaluation between multiple coverage criteria: combinatorial, structural and fault based. Our approach is tightly integrated with formal logic, since it uses formal logic to specify the system inputs (including the constraints), test predicates to formalize testing as a logic problem, and applies the SAL model checker tool to solve it, and hence to generate combinatorial test suites. Early results of experimental assessment are presented, supported by a prototype tool implementation.

1. INTRODUCTION

Verification of highly-configurable software systems, such as those supporting many optional or customizable features, is a challenging activity. In fact, due to its intrinsic complexity, formal specification of the whole system may require a great effort. Modeling activities may become extremely expensive and time consuming, and the tester may decide to model only the inputs and require they are sufficiently covered by tests. On the other hand, unintended interaction between optional features can lead to incorrect behaviors which may not be detected by traditional testing [27, 35]. To this aim, combinatorial interactive testing (CIT) techniques [12, 20, 27] can be effectively applied in practice [2, 31, 26]. CIT consists in employing combination strategies to select values for inputs and combine them to form test cases. The tests can then be used to check how the interaction among the inputs influences the behavior of the original system under test. The most used combinatorial testing approach is to systematically sample the set of inputs such a way that all t-way combinations of inputs are included. This approach exhaustively explores t-strength interaction between input parameters, generally in the smallest possible test executions.

In particular, pairwise interaction testing aims at generating a reduced-size test suite which covers all *pairs* of input values. Significant time savings can be achieved by implementing this kind of approach, as well as in general with t-wise interaction testing. As an example, exhaustive testing of a system with a hundred boolean configuration options would require 2^{100} test cases, while pairwise coverage for it can be accomplished with only 10 test cases. Similarly, pairwise coverage of a system with twenty ten-valued inputs (10^{20} distinct input assignments possible) requires a test suite sized less than 200 tests cases only. Also, it has been experimentally shown that CIT is really effective in revealing software defects [25]. A test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [32, 13]. Other experimental work shown that usually 100% of faults are already triggered by a relatively low degree of features interaction, typically 4-way to 6-way combinations [27], and that the testing of all pairwise interactions in a software system finds a significant percentage of the existing faults [13]. Dunietz *et al.* [15] compare t-wise coverage to random input testing with respect to structural (block) coverage achieved, with results showing higher reliability of the former in achieving block coverage if compared to random test suites of the same size. Burr and Young [6] report 93% code coverage as a result from applying pairwise testing of a commercial software system. For this reason combinatorial testing is used in practice and supported by many tools [29]. However, as explained in Section 2, most combinatorial testing techniques either ignore the constraints which the environment may impose on the inputs or require the user to modify the original specifications and add extra information to take into account the constraints. In this paper we investigate the use of CIT in the presence of constraints, and in particular with constraints over how input values can change over time, or briefly, temporal constraints. Our approach is particularly useful (but not limited to) if one wants to apply CIT to reactive systems, for which temporal constraints play a fundamental role.

We argue that a mixed approach, where both testing and formal analysis (model checking) tools are used in conjunction, could be of advantage in order to balance the required efforts over time. Specifically, we devise an approach where formal modeling of system's input/output domains and of its state space (behavior), is not required all at once but can be done in successive stages, respectively. At start-up stage, formal modeling of just the input domain allow for exhaus-

tive exploration of features interactions, using for instance combinatorial testing. This lets us achieve a high degree of confidence on the system correctness with relatively little effort. Meanwhile, or later in time, the same model can be extended to include the actual system's behavioral description. At this second stage, temporal properties which express constraints over sequences of inputs over time can also be checked. That is, feedback from the model checker can be used in order to customize the combinatorial test suites, allowing only valid tests, with respect to the requirements on the dynamics of the system parameters. Thus, still improving the significance of the resulting test process. Moreover, the behavioral model of the system can be also used as an oracle to compute expected outputs to each test, thus enabling also fully automated evaluation of the test process. In this context, we present a technique to express constraints over the dynamics of a system and to use them to build a valid combinatorial test suite. This technique has been implemented in a tool (ATGT)[1] designed in order to exploit model checkers to generate tests. Considering models possibly with their complete behavioral specification, allowed us to derive a combinatorial test suite and then evaluate its cross coverage with respect to structural and fault-based criteria.

The paper is organized as follows: section 2 gives some insight on the topic and recently published related works. Section 3 presents our approach, how we deal with propositional constraints, how we use the SAL model checker to generate combinatorial tests, and how we evaluate the tests. Section 4 explains how we incorporate temporal constraints over the input domain. Section 5 evaluates early results on some case studies carried out in order to assess the correctness of the proposed approach. Finally, section 6 draws our conclusions and points out some ideas for future extension of this work.

2. RELATED WORK

Many algorithms and tools for combinatorial interaction testing already exist in the literature. Grindal et al. count more than 40 papers and 10 strategies in their recent survey [20]. There is also a web site [29] devoted to this subject and several automatic tools are commercially [8] or freely available [32]. Most of the currently available methods and tools are strictly focused on providing an algorithmic solution to the mathematical problem of covering array generation only, while very few of them account also for other complementary features, which are rather important in order to make these methods really useful in practice in more general situations, like i.e. the ability to handle constraints on the input domains. In a previous paper [7] we have identified the following requirements for a effective combinatorial testing tool, extending the previous work on this topic by Lott et al. [28]:

A. Ability to deal with user specific requirements on the test suite. The user may require the explicit exclusion or inclusion of specific test cases, e.g. those generated by previous executions of the used tool or by any other means, in order to customize the resulting test suite. The tool could also let the user interactively guide the on-going test case selection process, step by step. Moreover the user may require the inclusion or exclusion of *sets of* test cases which refer to a particular critical scenario or combination of inputs. In this case the set is better described symbolically, for example by a predicate expression over the inputs. Note that *instant* [20] strategies, like algebraic constructions of orthogonal arrays and/or covering arrays, and *parameter-based*, iterative strategies, like IPO, do not allow this kind of interaction.

B. Integration with other testing techniques. Combinatorial testing is just *one* testing technique. The user may be interest to integrate results from many testing techniques, including those requiring very complex formalisms (as in [19, 18, 17, 16]). This shall not be limited to having a common user-interface for many tools. Instead, it should go in the direction of generating a unique test-suite which simultaneously accounts for multiple kinds of coverages (e.g., combinatorial, state, branch, faults, and so on). Our method, supported by a prototype tool, aims at bridging the gap between the need to formally prove any specific properties of a system, relying on a formal model for its description, and the need to also perform functional testing of its usage configurations, with a more accessible *black-box* approach based on efficient combinatorial test design. Integrating the use of a convenient model checker within a framework for pairwise interaction testing, our approach gives to the user the easy of having just one convenient and powerful formal approach for both uses.

C. Constraints support. A third desired requirement of a combinatorial testing strategy is the ability to deal with complex constraints. This issue has been recently investigated by Cohen et al. [9] and recognized as a highly desirable feature of a testing method. Note that the general problem of finding a minimal set of test cases that satisfies t-wise coverage can be NP-complete [34, 30]. If constraints on the input domain are to be taken into account, even the generation of a single test can be NP-complete, since it can be reduced in the most general case to a satisfiability problem.

Although no one has considered constraints over the evolution of monitored variables during the time, there are already few approaches to deal with the non temporal (or propositional) constraints over the inputs.

In order to deal with constraints, some methods require to remodel the original specification. For instance, AETG [8, 28] requires to separate the inputs in a way they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [9]) can be directly modeled in the specification. Other methods [21] require to explicitly list all the forbidden combinations. As the number of input grows, the explicit list may explode. In [3] the authors introduce the concept of *soft constraints*: they use a method to avoid tuples if possible. In this paper we consider only hard constraints: a test is valid only if it satisfies the constraints. Cohen et al. [9] found that just one tool, PICT [11], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each

[1]ATGT tool is available for download at: http://cs.unibg.it/gargantini/projects/atgt.

forbidden test cases. However, there is no detail on how the constraints are actually implemented in PICT, limiting the reuse of its technique.

Cohen et al. [9] propose a framework to incorporating constraints into established greedy and simulating annealing combinatorial testing algorithm. Their framework is general and fully supports the presence of constraints, even if they can be modeled only as forbidden tuples.

Recently, several papers investigated the use of verification methods for combinatorial testing. Hnich et al. [24] translates the problem of building covering arrays to a Boolean satisfiability problem and then they use a SAT solver to generate their solution. In their paper, they leave the treatment of auxiliary constraints over the inputs as future work. Conversely, Cohen et al. Kuhn and Okun [25] try to integrate combinatorial testing with model checking (SMV) to provide automated specification based testing, with no support for constraints.

In our previous work [7] we have investigated the integration of model checkers with combinatorial testing in the presence of (propositional) constraints while supporting all of the additional features listed above. In [7], not only we address the use of full constraints as suggested in [9] but we feature the use of predicates to express constraints over the inputs (see section 3 for details). Furthermore, while Cohen's general constraints representation strategy has to be integrated with an external tool for combinatorial testing, our approach tackles every aspect of the test suite generation process.

In this paper we extend and integrate our previous work with some modifications by considering t-wise coverage, dealing with temporal constraints over the dynamics of inputs, which has not yet been investigated, to the best of our knowledge, and evaluating combinatorial tests against structural and fault-based test suites (cross coverage evaluation).

3. LOGIC-BASED APPROACH

In this section we present the logic-based approach presented in [7] with some integrations and extensions, like the n-wise coverage. The technique is supported by the *ASM Test Generation Tool* (ATGT). ATGT was originally developed to support structural [19] and fault based testing [16] of *Abstract State Machines* (ASMs), and it has been extended to support also combinatorial testing.

Since pairwise testing aims at validating each possible pair of input values for a given system under test, we then formally express each pair as a corresponding logical expression, a *test predicate* (or test goal), e.g.:

$$p_1 = v_1 \wedge p_2 = v_2$$

where p_1 and p_2 are two inputs or monitored variables of enumerative or boolean domain and v_1 and v_2 are two possible values of $p1$ and $p2$ respectively. Similarly, the n-wise coverage can be modeled by a set of test predicates, each of the type:

$$p_1 = v_1 \wedge p_2 = v_2 \wedge \cdots \wedge p_n = v_n$$

where $p_1, p_2 \ldots p_n$ are n inputs and $v_1, v_2 \ldots v_n$ are their values, such that every possible combination of the n input variables with their values is taken into account. Please note that to reach complete n-wise coverage this has to be true for each n-tuple of input parameters of the considered system.

The easiest way to enumerate the test predicates required for the n-wise coverage of an ASM model is to employ a combinatorial enumeration algorithm, which simply loops over the variables and their values to build all the possible test predicates.

In order to correctly generate the test predicates required by the coverage we assume the availability of a formal description of the system under test. This description should include at least the input parameter domains[2]. The description has to be entered in the tool as an ASM specification in the AsmetaL language [33]. We use as case study the well known example Cruise Control (CC) [1], whose AsmetaL specification in shown in Listing 1. The CC has 4 boolean monitored variables, one monitored variable with 3 possible values, and, for instance, the collection of test predicate for the pairwise coverage count 48 predicates. These are the combinatorial explosion of all assignments for each of the five possible subsets of two distinct parameters of CC. The the four-wise coverage set for the same example count 112 test predicates. They can be obtained by enumerating all the possible assignments for the following parameter subsets:

fast	igOn	brake	engRun
fast	igOn	brake	lever
fast	igOn	engRun	lever
fast	brake	engRun	lever
igOn	brake	engRun	lever

Table 1: parameters combinations

This activity (step 1) is carried out by the *test predicate generator* of Fig. 1 showing the process proposed by our method and implemented in ATGT.

By formalizing the n-wise testing by means of logical predicates, finding a test that satisfies a given predicate reduces to a logical problem of finding a complete [3] model for a logical formula. To this aim, many techniques like constraint solvers, can be applied. Our approach exploits a well known model checker tool, namely the bounded and symbolic model checker tool SAL [14]. Given a test predicate *tp*, SAL is asked to verify a *trap property* [17] which states that *tp* is never true, or $never(tp)$, which in LTL, the language of SAL, becomes $G(\neg tp)$. The *trap property* is not a real system property, but enforces the generation of a counter example, that is a set of assignments falsifying the trap property and satisfying our test predicate. The counter example will contain bindings for all monitored inputs, including those parameters missing (free) in the predicate, thus defining the test we were looking for.

[2]Currently, only finite, discrete enumerable domains are supported.

[3]We say that a model is *complete* if it assigns a value to every input variable. We informally call this model *assignment*

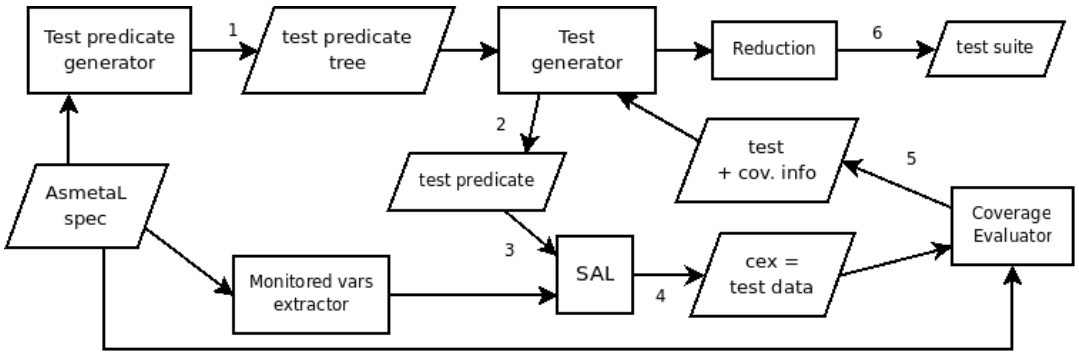

Figure 1: the process of test suite generation based on SAL MC

```
asm cruiseControl
import StandardLibrary
//UNIVERSES and FUNCTIONS
signature:
  enum domain CCMode =
        {OFF| INACTIVE|CRUISE|OVERRIDE}
  enum domain CCLever =
        {DEACTIVATE| ACTIVATE|RESUME}
  dynamic controlled mode : CCMode
  dynamic monitored lever : CCLever
  dynamic monitored igOn : Boolean
  dynamic monitored engRun : Boolean
  dynamic monitored brake : Boolean
  dynamic monitored fast : Boolean
definitions:
// AXIOMS: ADDED LATER
// RULES:
main rule r_CruiseControl =
 if not igOn then mode := OFF
 else if not engRun then mode:= INACTIVE
 // igOn and engRun
 else par
  if mode = OFF then mode := INACTIVE endif
  if mode = INACTIVE and not brake and not fast
      and lever = ACTIVATE then
        mode := CRUISE
  endif
  if mode = CRUISE then
    if fast then mode := INACTIVE
    else if brake or lever = DEACTIVATE then
                    mode := OVERRIDE endif endif
  endif
  if mode = OVERRIDE and not fast and not brake
      and (lever = ACTIVATE or lever = RESUME) then
        mode := CRUISE
  endif endpar
endif endif

default init s1:
// INITIAL STATE: ADDED LATER
```

Listing 1: AsmetaL specification of Cruise Control

```
cruiseControl: CONTEXT = BEGIN

 CCLever : TYPE = {DEACTIVATE, ACTIVATE, RESUME};

  monitored : MODULE = BEGIN

  OUTPUT igOn, fast, engRun, brake: BOOLEAN,
         lever: CCLever

  TRANSITION igOn' IN {true, false};
       fast' IN {true, false};
       engRun' IN {true, false};
       brake' IN {true, false};
       lever' IN {DEACTIVATE, ACTIVATE, RESUME};
 END;

 % trap property
 tc_92668c : THEOREM monitored |- G(NOT <tp>);
END
```

Listing 2: SAL specification of Cruise Control

The steps we actually perform to generate a suitable test suite are depicted in Fig. 1. We randomly extract a test predicate *tp* (step 2) from the set of all the test predicates previously generated. The user may select only a subset of the test predicates for generation or include some extra tps (as explained in [7]): we call *candidates* all the test predicates to be considered. Then (step 3) we build the SAL specification by considering the inputs of the original model and the trap property. The SAL translation of CC is shown in Listing 2.

We run SAL to obtain a counter example, i.e. an assignment of every input which satisfies *tp*. Without constraints such counter example always exists, it represents the test, and it is called *test data* (step 4). The test data produced by SAL is then completed to compute the expected values for the controlled variables to obtain a real test. Indeed, since the SAL model ignores the rules and the controlled variables and it considers only the inputs and their domains, the counter example does not contain the expected values for the controlled variables. The test is also evaluated to check if it covers other candidates, i.e. if it satisfies other test predi-

T = test suite to be optimized
Op = optimized test suite
Tp = set of test predicates which are not covered by tests in Op

0. set Op to the empty set and add to Tp all the test predicates
1. take the test t in T which covers most test predicates in Tp and add t to Op
2. remove all the test predicates covered by t from Tp
3. if Tp is empty then return Op else goto 1

Figure 2: Test suite reduction algorithm

cates (step 5). Finally the test is added to the test suite and the test predicates covered are removed from the candidates until the set becomes empty. This approach, according to [20], can be classified as iterative, since the test suite is built one test at the time.

Even if one skips the test predicates already covered, the final test suite may still contain some test cases which are redundant. We say that a test case is *required* if contains at least a test predicate not already covered by other test cases in the test suite. We than try to reduce the test suite by deleting all the test cases which are not required in order to obtain a final test suite with fewer test cases. Note, however, that an unnecessary test case may become necessary after deleting another test case from the test suite, hence we cannot simply remove all the unnecessary test predicates at once. We have implemented a greedy algorithm, reported in Fig. 2, which finds a test suite with the minimum number of required test cases by simply looking at which test predicates are covered by each test in the original test suite. This reduction technique is applied in step 6 of the Fig. 1.

3.1 Propositional constraints

Support for constraints over the inputs is given by expressing them as axioms in the specification. In the CC example, the assumptions that the engine is running only if the ignition is on and that the car is driving too fast only if the engine is running, are modeled in AsmetaL by the following axioms:

axiom inv_ignition **over** engRun : (engRun implies igOn)
axiom inv_toofast **over** fast : (fast implies engRun)

To express constraints we adopt the language of propositional logic with equality[4]. Note that most methods and tools admit only few templates for constraints: the translation of those templates in equality logic is straightforward. For example the **require** constraint is translated to an *implication*; the **not supported** to a *not*, and so on. Even the method proposed in [9] which adopt a similar approach to ours prefer to allow constraints only in a form of forbidden configurations [22], since it relies for the actual tests generation on external tools. Our approach allows the designer to state the constraint of a forbidden combination as a *not* statement. Moreover, we support constraint that not only relate two variable values (to exclude a pair), but can contain generic bindings among variables. Note that any constraint models an explicit binding, but their combination may give rise to complex implicit constraints.

In our approach, the axioms must be satisfied by any test case we obtain from the specification, i.e. a test case is *valid* only if it does not contradict any axiom in the specification. While others [4] distinguish between forbidden combinations and combinations to be avoided, we consider only forbidden combinations, i.e. combinations which do not satisfy the axioms. Since we allow the specification to contain also controlled variables and rules that assign value to them, error conditions can be modeled by an *error* controlled variable, and rules that detect erroneous conditions and assign suitable values to *error* in order to signal the occurrence of such conditions. The specification can be used then as oracle to know whether a combination causes an error in the system.

In the presence of constraints, finding a valid test case becomes a challenge similar to finding a counter example for a theorem or proving it. Verification techniques like SAT algorithms, or model checkers algorithms are particularly effective in this case, so we investigated the use of the bounded and symbolic model checkers in SAL to this aim. To include constraints in SAL they must be translated in order to embed the axioms directly in the trap property, since SAL does not support assumptions directly. Simply put, the trap property must be modified to take into account the axioms $a_1, a_2, ...a_n$. The general schema for it becomes:

$$G(a_1 \wedge a_2 \wedge .. \wedge a_n) \Rightarrow G(\neg tp) \quad (1)$$

A counter example of the trap property 1 is still a valid test data. In fact, if the model checker finds an assignment to the variables that makes the trap property false, it finds a case in which both the axioms are true and the implied part of the trap property is false. This test case covers the test predicate and satisfies the constraints.

Without constraints, we were sure that a trap property derived from a consistent test predicate had always a counter example. Now, due to the constraints, the trap property (1) may not have a counter example, i.e. it could be true and hence provable by the model checker. We can distinguish two cases. The simplest case is when the axioms are inconsistent, i.e. there is no assignment that can satisfy all the constraints. In this case each trap property is trivially true since the first part of the implication (1) is always false. The inconsistency may be not easily discovered by hand, since the axioms give rise to some implicit constraints, whose consequences are not immediately detected by human inspection. For example a constraint may require $a \neq x$, another $b \neq y$ while another requires $a \neq x \rightarrow b = y$; these constraints are inconsistent since there is no test case that can satisfy them. Note that also input domains must be taken into account when checking axioms consistency. Inconsistent axioms must be considered as a fault in the specification and this must be detected and eliminated. For this reason when we start the generation of tests, if the specifications has axioms, we check that the axioms are consistent by trying to prove:

[4]To be more precise, we use propositional calculus, boolean and enumerative types for variables, and equality

$$G(\neg(a_1 \wedge a_2 \wedge .. \wedge a_n)) \quad (2)$$

If this is proved by the model checker, then we warn the user, who can ignore this warning and proceed to generate tests, but no test will be generated, since no valid test case can be found. We assume now that the axioms are consistent. Even with consistent axioms, some (but not all) trap properties can be true: there is no test case that can satisfy the test predicate and the constraints. In this case we define the test predicate as *infeasible*.

Definition 1. *Let tp a test predicate, M the specification, and C the conjunction of all the axioms. If the axioms are consistent and the trap property for tp is true, i.e.* $M \wedge C \models \neg tp$*, then we say that tp is infeasible. Let tp be the n-wise test predicate* $p_1 = v_1 \wedge p_2 = v_2 \ldots p_n = v_n$*, we say that this combination of assignments is infeasible.*

An infeasible combination of assignments represents a set of invalid test cases: all the test cases which contain this combination are invalid. Our method is able to detect infeasible assignments, since it can actually prove the trap property derived from it. The tool finds and marks the infeasible combinations, and the user may derive from them invalid tests to test the fault tolerance of the system. For example, the following test predicate results infeasible for the CC example, since the engine cannot run when the ignition is off:

engRun = true AND igOn = false −−−> unfeasible

Note that since the BMC is in general not able to prove a theorem, but only to find counter examples, it would be not suitable to prove infeasibility of test predicates. However, since we know that if the counter example exists then it has length (i.e. the number of system states) equal to 1, if the BMC does not find it then we can infer that the test predicate is infeasible. Note that a test specifies the exact value of all the input variables, while a test predicate specifies a generic scenario. ATGT allows the tester to load an external file containing user defined tests and test goals. When an external file is loaded, ATGT adds the user defined test in the set of test predicates to be covered. Then it adds the user defined tests and it checks which test predicates are satisfied by these tests. In this way the tester can decide to skip the test predicates covered by tests he/she has written ad hoc.

4. TEMPORAL CONSTRAINTS

Until now, we have considered only constraints which bind the input values at the same time, e.g. a variable cannot have a value if another has another value. Now we consider systems which evolve during their operations in a discrete way, step by step. At every step every variable can have a different value from the value it had before. In particular, the monitored variables are free to non deterministically change from one step to the next one. However, some axioms may limit how inputs evolves due some external constraints of the environment. For these systems, a test is no longer a simple assignment to every input variable to one of its possible values, but it becomes a sequence of assignments, where every assignment denotes a state. In this case we refer to a test as *test sequence*. We consider in this paper three kinds of temporal constraints: *initial value*, *next value*, and *one input assumption* (OIA). These three kinds of constraints are the typical types of assumptions most formal notation for reactive systems permit about the models under test. The OIA is useful to model asynchronous systems, which process an input event at the time.

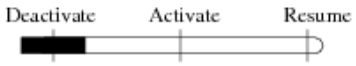

Figure 3: the lever of a Cruise Control

Initial and next value. The first two constraints state how a *single* input can evolve during the normal operation of the system. Consider for example the lever for a cruise control system depicted in Fig. 3. The lever is initially in the DEACTIVATE position and from that position it can only become ACTIVATE. From the ACTIVATE position it may become RESUME or DEACTIVATE again.

If the tester wants to use the test generated by our method to test the actual system for conformance with the requirements, he/she must use the actual interface to enter the input values of the test cases. For example, if a test case requires that lever = RESUME, the user must start with lever = DEACTIVATE, than lever = ACTIVATE and finally lever = RESUME. The application of *one* test case has required three steps, but the testing method considered so far is not aware of this and it cannot take advantage: if another test requires the user to switch lever = ACTIVATE, the tester must go back to the initial state and start again the application of the new test case, although this kind of combination of inputs has already been tested. To consider this kind of constraints we extend our method as follows. First, the tester will add the constraints in the specification. Initial values are set by the following initialization in the AsmetaL specification.

```
default init s1:
  function mode = OFF
  function lever = DEACTIVATE
  function igOn = false
  function engRun= false
  function brake = false
  function fast = false
```

The next value constraint can be specified by using the special library function *next*. For example, for CC a constraint is:

```
axiom lever_deact : lever = DEACTIVATE implies
  next(lever = DEACTIVATE) or next(lever = ACTIVATE)
```

While the initial values are translated into SAL by the INITIALIZATION clause, the next value constraints are translated in SAL in *Linear Temporal Logic* (LTL)[5]. The trap

[5]The SAL model checkers use LTL (Linear Temporal Logic)

property 1 will contain in $a_1 \ldots a_n$ not only the propositional axioms, but also the temporal axioms. For example, to express the constraint when the lever is in the DEACTIVATE position, the trap property will contain the following axiom:

```
G( ... AND lever = DEACTIVATE =>
  (X(lever = DEACTIVATE) OR X(lever = ACTIVATE))) ....
```

One Input Assumption. The One Input Assumption constraint, taken form [23], allows at most one monitored variable to change from one state to the next. The expression in AsmetaL of such constraint in terms of next values is very complex and it can be error prone. We decided to allow the user to simply set this constraint as a preference of the generation method. The translation in SAL differs from the next constraints too: instead adding a complex axiom, we modify the TRANSITION clause in the SAL specification as follows, thus allowing only one input to change at the time:

```
TRANSITION
  [ true --> igOn' IN {true, false};
  [] true --> fast' IN {true, false};
...
  ]
```

In the presence of temporal constraints, the counter example produced by SAL will contain several states. We extend the definition of test: a **test data sequence** is a sequence of assignments, whose first assignment is compatible with the initial value constraint and every pair of consecutive assignments is compatible with the temporal constraints.

We have now to take the whole counter example produced by the model checker as a test sequence: a test case is now a test data sequence with a possible number of states greater than 1. Furthermore, adding this kind of constraints limits (but it does not exclude) the use of the Bounded Model Checker (and any other constraint solver or SAT based technique). Indeed the BMC is no longer able to prove that a test predicate is infeasible, since not finding a counter example of a given finite length does not necessarily imply that the trap property is true: it can be the case that the counter example is longer than the BMC depth. We are investigating the use of induction and a conservative use of the bound for counter examples to be able to induce infeasibility from the non existence of a counter example by using BMC. Meanwhile, if the BMC does not find a counter example for a test predicate, we do not mark that test predicate as unfeasible, we warn the user, and we invite to run the SMC.

5. EXPERIMENTS

We have applied our technique to two case studies: the Cruise Control (CC) , the Safety Injection System (SIS), a simplified version of the system described in [10]. Table 2 reports some significant data about them. The *input size* is the product of the input domain sizes. The notation n^m means that the specification contains m variables each with n possible values. The table reports the number of controlled variables (# C), the number of rules (# R), the number of axioms (#*axs*: propositional and temporal constraints, which include for SIS the OIA), and the number of the test predicates obtained from the n-wise combinatorial coverage. For CC some test predicates were proved unfeasible.

as their assertion language but they also accept CTL syntax on the common fragment

	input size	# *C*	# *R*	# *axs*	*#tp n-wise* 2	3	4	5
CC	$2^4 \cdot 3$	1	6	5	48	104	112	48
	unfeasible				3	19	38	24
SIS	$2^2 \cdot 3$	3	4	4	16	12	-	-

Table 2: Case Studies

The first goal of our experiments has been the validation of our technique by generating the test suites for the combinatorial testing of the two case studies. We have used the two different model checkers SAL provides: the symbolic model checker (smc) and the bounded model checker (bmc). We wanted also to study the effects of the constraints over the test suites. We have modified the original specifications in order to obtain two versions: one with the constraints (denoted by +) and one without (temporal and propositional) constraints (denoted by -). Table 3 reports the total time required to generate the test suite (A) for each n-wise covering test suite, the total number of tests in the suite, the total number of states (as sum of the number of states in all the test sequences - *#sts*) before and after the reduction algorithm is applied (if the reduction has actually reduced the size of the test suite). The column (B) shows also the average time taken for every test in the test suite and it is equal to (A)/#tests. Note that the number of tests and the number of states are equal for specifications without constraints since each test has only one state.

Observing the data in Table 3 we can make the following preliminary observations. The BMC proved to be faster than the SMC, especially in the presence of constraints. The total time to generate the test suite (column A) is always lower for the BMC than for the SMC, while the average time taken per every single generated test (column B) is always lower for the BMC in the presence of constraints, while it is comparable without constraints. The number of tests and the total length is again lower for the BMC than for the SMC (without constraints the sizes are comparable again). This suggests that the BMC is more suitable to deal with constraints than the SMC: it is faster and produces smaller test suites. This is particularly true in case of short tests as in our case studies. However, the average length of the tests (not reported in Table) produced by the SMC is lower: since the counter example produced by the SMC is as short as possible, the SMC produces very short tests, which cover only few test predicates. This, on the other hand, causes the SMC to run more times than the BMC and the start-up time for SMC is longer than the BMC since it must build the complete BDD representation of the problem. SMC works better when one needs very short tests while the number of tests is not so important. The reduction technique reduced the test suite only in a few cases (column *after red.*).

The constraints caused the number of tests to decrease, but the total number of states to increase. We never found a test

		(A)	no red.		*after red.*		(B)
n-wise	mc	time secs	# tests	# sts	# tests	# sts	time x test
CC + (with constraints)							
pair	smc	10.4	6	27	5	22	1.73
	bmc	5.2	4	27	-	-	1.3
3	smc	30.9	14	58	13	53	2.21
	bmc	6.3	5	46	4	39	1.26
4	smc	50.2	19	77	-	-	2.64
	bmc	10.5	8	72	-	-	1.31
5	smc	45.2	21	83	20	81	2.15
	bmc	10.4	8	81	-	-	1.3
CC - (without constraints)							
pair	smc	18.8	21	21	-	-	0.9
	bmc	18.9	21	21	-	-	0.9
3	smc	34.8	37	37	-	-	0.94
	bmc	33.7	37	37	-	-	0.91
4	smc	41.9	46	46	-	-	0.91
	bmc	41.9	46	46	-	-	0.91
5	smc	44.5	48	48	-	-	0.93
	bmc	45.3	48	48	-	-	0.94
SIS + (with constraints)							
pair	smc	5.5	6	29	-	-	0.92
	bmc	2.5	3	19	-	-	0.83
3	smc	4.4	5	28	-	-	0.88
	bmc	3.3	4	33	-	-	0.83
SIS - (without constraints)							
pair	smc	4.8	10	10	-	-	0.48
	bmc	5.5	10	10	-	-	0.55
3	smc	6.0	12	12	-	-	0.5
	bmc	6.2	12	12	-	-	0.52

Table 3: Test suite with and without constraints

suite complaint with the constraints able to cover all the test predicates in a total number of states equal to the number of states of the test suite without constraints. Although considering the temporal constraints notably increases the total length of the test suite, we believe that it augments the usability and applicability of the test suite, as observed in Sect. 4.

5.1 Cross coverage evaluation

We have compared the coverage obtained by the combinatorial n-wise testing with the coverage obtained by the structural criteria presented in [18] and the fault based criteria presented in [16]. The results for the CC and SIS examples are reported in Tables 4 and 5, where BR is the basic rule coverage (similar to the branch coverage), CR is the complete rule coverage, UR is the update rule coverage, MCDC is the modified condition decision coverage, ASF is the associative shift fault, ENF is the expression negation fault, LNF is the literal negation fault, ORF is the (logical) operator reference fault, ST0 is stuck at false, ST1 is stuck at true, and ROF is the relational operator fault.

Table 4 shows the number of (feasible) test predicates for all the structural and fault based coverage criteria (row *#tp*) and the number of these test predicates covered by each n-wise combinatorial test suite.

The table shows that the pairwise coverage implied a very low coverage of structural and fault based criteria for the CC example. Small improvements were obtained by increasing n of the the n-wise coverage. However, combinatorial testing was not even able to cover all the rules (BR), although the number of combinatorial test predicates is much higher than the number of test predicates for BR. We have investigated and found that the rules contain guards like the following one (see Listing 1):

if mode = INACTIVE and not brake and
not fast and lever = ACTIVATE **then** ...

which requires to be covered a particular value of the controlled variable *mode* and a particular combination of inputs. The desired input combination is covered in at least a test in the test suite, but since combinatorial testing ignores the outputs, such combination may not activate that particular rule because the controlled variable has a different value from that desired. By ignoring the controlled variables, combinatorial testing may be not able to drive the system to a critical state where a controlled variable takes a particular value. These results prove that the combinatorial testing does not always imply good structural coverage, contrary to the experiments presented in [6, 15], where, however, the structural coverage was computed against the implementations and not against the specifications as in our approach. Our experiments would confirm the results presented in [5], where the interaction test suites provided little benefit over the randomly generated tests and did not improve coverage of the requirements-based tests. On the contrary, we obtained that the combinatorial coverage implies a very high structural and fault based coverage for the SIS, as reported again in Table 4. We observed that the rules in SIS were not so dependent on controlled variables as those in CC, and this is the reason why we obtained better structural coverage for SIS than for CC. These results raise concerns on the application of combinatorial testing in the model-based domain for embedded and reactive systems, where the information about the system state is needed to build test suites achieving good structural coverage. However, the combinatorial testing may be the only model-based testing technique applicable in case the model contains only the specification of the inputs together with their domains and constraints.

On the other hand, structural and fault based criteria do not imply combinatorial testing either, as reported in Table 5, which shows the number of combinatorial test predicates covered by structural and fault based testing. For this reason we believe that combinatorial, structural and fault based criteria are complementary each other.

6. CONCLUSION AND FUTURE WORK

In this paper we have described an integrated approach to use formal analysis in conjunction with traditional testing in order to improve the efficiency of the verification&validation process. We presented an approach which is tightly integrated with formal logic, since it uses a formal notation to specify the system under test, test predicates to formalize combinatorial testing as a logic problem, and applies a model checker to solve it. We developed and presented a technique for the construction of n-wise combinatorial test suites, featuring not just simple constraints over the set of inputs but

		Structural Coverage				Fault Based							
		BR	CR	UR	MCDC	ASF	ENF	LNF	MLF	ORF	ST0	ST1	ROF
CC	*#tp*	*7*		*7*	*9*	*0*	*22*	*29*	*28*	*22*	*49*	*49*	*29*
	pairwise	3		4	5		19	11	9	19	10	28	5
	3-wise	3		3	5		19	12	10	19	10	29	5
	4-wise	3		3	5		19	13	11	19	10	30	6
	5-wise	4		4	6		19	15	11	19	20	31	6
SIS	*#tp*	*7*	*2*	*11*	*7*	*1*	*9*	*16*	*16*	*9*	*24*	*24*	*16*
	pairwise	7	2	9	7	1	9	16	15	9	24	21	15
	3-wise	7	2	10	7	1	9	16	16	9	24	24	15

Table 4: Cross coverage of n-wise combinatorial testing

			Structural Coverage				Fault Based							
		#tp	BR	CR	UR	MCDC	ASF	ENF	LNF	MLF	ORF	ST0	ST1	ROF
CC	pairwise	*45*	26	-	26	26		14	31	40	40	40	40	40
	3-wise	*85*	34	-	34	34		16	48	64	64	64	64	64
	4-wise	*74*	21	-	21	21		9	32	47	47	47	47	47
	5-wise	*24*	5	-	5	5		2	7	13	13	13	13	13
SIS	pairwise	*16*	9	5	10	9	6	3	9	10	10	10	10	10
	3-wise	*12*	4	2	5	4	2	1	4	5	5	5	5	5

Table 5: Combinatorial test predicates covered by structural and fault based testing

also over the evolution of inputs, which is the major and original contribution. Moreover, we were able to evaluate the cross coverage between combinatorial, structural and fault based coverage criteria, with very limited effort. Early results of experimental assessment have also been presented, supported by a prototype tool implementation. We found some interesting and unexpected results, since in one case combinatorial testing did not imply structural testing, in spite of different results in the literature suggest the contrary. We believe the whole proposed approach can be very successful since it still allows the tester/developer to interface to an easy black-box test tool - where only combinatorial testing can be used - and, at the same time, advantage of the (available) formal specification in order to implement a more effective test process - where structural and fault based testing can be applied.

Since our approach is based on the use of model checkers, it suffers from the state explosion problem. However, for the combinatorial testing, we consider only the inputs together with their constraints and this should keep the problem tractable. Other approaches may take advantage of a limited expressiveness of the constraints language, but with a loss of usability. We plan to investigate the combined use of the model checker for the constrained part of the model with classical algorithms for combinatorial testing for the unconstrained part to minimize the total complexity of the method.

7. REFERENCES

[1] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *International Symposium on Software Testing and Analysis*. ACM, 1996.

[2] R. Brownlie, J.Prowse, and M. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.

[3] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.

[4] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.

[5] R. C. Bryce, A. Rajan, and M. P. E. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 259–268. IEEE Computer Society, 2006.

[6] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.

[7] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2008.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7), 1997.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.

[10] P.-J. Courtois and D. L. Parnas. Documentation for

safety critical software. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 315–323. IEEE Computer Society Press, May 1993.

[11] J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.

[12] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott. Model-based testing of a highly programmable system. *issre*, 00:174, 1998.

[13] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering ICSE*, pages 285–295, New York, May 1999. Association for Computing Machinery.

[14] L. de Moura, S. Owre, H. Rueß, J. R. N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[15] I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. Mallows, and A. Iannino. Applying design of experiments to software testing. In I. Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.

[16] A. Gargantini. Using model checking to generate fault detecting tests. In *International Conference on Tests And Proofs (TAP)*, number 4454 in Lecture Notes in Computer Science (LNCS), pages 189–206. Springer, 2007.

[17] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99*, number 1687 in LNCS, 1999.

[18] A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS, Volume 10 Number 8 (Nov 2001)*, 2001.

[19] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from ASM specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.

[20] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[21] A. Hartman. Ibm intelligent test case handler: Whitch, http://www.alphaworks.ibm.com/tech/whitch.

[22] A. Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.

[23] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[24] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[25] D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. In *SEW '06: IEEE/NASA Software Engineering Workshop*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[26] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In I. Society, editor, *27th NASA/IEEE Software Engineering workshop*, pages 91–95, 2002.

[27] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.

[28] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[29] Pairwise web site. http://www.pairwise.org/.

[30] G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.

[31] B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In C. Breckenridge, editor, *Proceedings of the Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.

[32] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[33] The ASMETA project. http://asmeta.sourceforge.net.

[34] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom) Berlin, Germany*, pages 283–298, march 2002.

[35] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.

A Case Study in Automated, Modular, and Full Functional Verification

Jason Kirschenbaum
The Ohio State University
Columbus, OH 43210, USA
kirschen@cse.ohio-state.edu

Heather Harton
Clemson University
Clemson, S.C. 29634, USA
hkeown@clemson.edu

Murali Sitaraman
Clemson University
Clemson, S.C. 29634, USA
murali@cs.clemson.edu

ABSTRACT

Mechanical and full verification of behavior of object-based programs is a central software engineering problem. Any successful solution to the problem should strike a delicate compromise between amenability to automation and several software engineering factors, such as the form and ease of specifications, demands on software developers to provide invariants and hints, development and use of relevant mathematical theories, and language and software design. The object of this paper is to illustrate the issues that need to be addressed for full behavioral verification through our experiments towards push-button verification of an imperative object-based code in a modular fashion. In the process, the case study indirectly characterizes the requirements of a language for developing verifiable software.

1. INTRODUCTION

Automated verification of component-based software is a difficult and challenging problem, and it needs to be tackled for the verification grand challenge [8, 21] to succeed. The present paper illustrates the issues in automating verification using object-based sorting as a case study.

This case study differs from previous efforts in automated verification of properties in that the focus is on full behavioral verification. The verification approach used here is clean [10], because it uses both a language and specifications that do not involve modeling references or aliasing. It also differs from the work on automated verification of sorting algorithms (e.g. [1]) in several respects. The verification approach is modular; it is based only on the specification of components that are reused and allows verification of one component at a time. The sorting specification and the verified code are both generic. The proof of correctness is total. Finally, the mathematical notions come from a user-defined library illustrating an open system for verification of non-trivial software. Overall, a key contribution of the case study is in indirectly characterizing the requirements of an integrated language for mathematical specification and implementation as a first step towards automating verification.

By design, the chosen example software is simple and is in the same spirit as the set of benchmarks for automatic verification that we have proposed in a recent paper [19]. However, while the example is simple, the issues raised by this code must be dealt with by any verification approach. Even with a simple example, our attempt to achieve push-button verification of the software quickly ran into several difficulties, demonstrating a variety of issues that need to be addressed for automated verification to become practical. The case study is a "real" example in the sense that it was chosen for verification from the collection of components available at the OSU RSRG software components library [2]. Furthermore, the code used in the case study predates the present verification effort. Though the component had been engineered with an eye toward verification, automating this task has proved challenging.

The first of the challenges in our case study, likely typical of other such efforts, concerns human errors in writing assertions. That humans will have to write mathematical assertions, such as to specify the behavior of software they intend to verify, is an inescapable requirement of any attempt to produce correct software [18]. There was a subtle error in the specification from the use of an output value of a variable instead of its input value in an assertion. Even after that error was fixed, one of the sub-problems in establishing an automated proof had to do with a weak invariant for a loop; while valid and apparently adequate, the invariant was not sufficiently strong to complete the proof. What is noteworthy here is that these errors existed in a piece of software that has been studied (and used) by ourselves, other instructors, and literally thousands of students. This observation also underscores why "social proofs" [14] are necessarily subject to errors implying that complete automation, the focus of this workshop, is absolutely essential.

The second set of challenges concern difficulties in achieving automation. Our proof process is based on [7] and the verification conditions (VC) resulting from our VC generator are fed to the Isabelle proof assistant [15]. Even after human errors in writing specification and invariants were eliminated, automation using the Isabelle proof assistant as the back-end prover remained elusive. The problem was traced to insufficient theory development and the inability to instantiate a universal quantifier suitably without any human help. Our attempt to tackle these challenges has led to a novel way of using suitable definitions and theory devel-

opment to minimize and avoid, if possible, the explicit need for universal quantifiers. It is too early to tell whether the proposed approach will generalize or whether it would simply raise problems elsewhere that are not apparent in our current setting.

It is important that we note that our choice of Isabelle, which is not a totally automated prover, is motivated by several factors. While dedicated first-order theorem provers might be able to automate more given the right axioms and background facts, we need a higher order prover. Isabelle also allows us to call external theorem provers (e.g. [17, 20]). Furthermore, we want our verification to be performed using verified theories; Isabelle has one of the largest libraries of proven mathematical theorems. This is especially important in full verification where behaviors of objects may be specified using sophisticated mathematical models. The second factor is that Isabelle has support for both automated and user-guided proof methods. This is especially useful to experimental researchers, because that support helps us find where a proof of a VC gets stuck, find a correct proof, and use that information to modify the automated proof methods (thereby increasing the types of VCs that are provable automatically). Finally, we note that we know of no provers that can automatically discharge the VCs from our case study.

In attempting to solve the subproblems during automation, we also generalized our mathematical definitions and software specifications in ways that we hadn't initially conceived. The resulting software is clearly more reusable. This last point may indicate that software designed for automated verification might be, in general, better engineered than other software and that automation has benefits, not just for correctness, but also for the overall quality of software.

The methods and tools used for the subject of this paper have been set up such that they can be repeated elsewhere with other scenarios and case studies. We include only partial code and proofs in this paper. Complete details may be found at: `www.cs.clemson.edu/~resolve/benchmarks`.

2. CASE STUDY IN DETAIL

```
Begin String_Theory

Definition String( Str(G : Set) : Set,
                   empty_string : Str(G),
                   ext( a : Str(G), x : G): B =

...

Inductive Definition on b : Str(G : Set)
                 of (a : Str(G)) o (b : Str(G)) is
...

Lemma ConcatenationAssociative
 a o (b o c) = (a o b) o c

Definition IsPermutation where
 IsPermutation a b = ...

Lemma PermutationCommutative
 IsPermutation (x o y) (y o x)

...
End String_Theory
```

Figure 1: Example of String Theory

A specification and implementation of a sort operation that extends a Queue abstract data type is given below in the RESOLVE [16, 4, 3, 11] notation. Here, queues are conceptualized mathematically as strings of entries, constrained to be within a given Max_Length. The specification of the sort operation is parameterized both by the type of the entries the sorted queue contains and by the ordering relation used for sorting. A sample of the theory is given in Figure 1. The proofs of the lemmas and theorems in the theory are given in a separate proof module which is not shown.

Operation Sort_Q has no preconditions. It ensures that the Queue parameter Q after the operation must be nondecreasing (with respect to the client supplied LEQV relation) and must be a permutation of the incoming Q (denoted by #Q).

In the figure, the local definition IsNondecreasing is based on the the caller-supplied LEQV relation. It uses "o" to denote string concatenation. The definition of IsPermutation is in String_Theory a mathematical unit imported by the specification Queue_Template neither of which is shown.

```
Enhancement Sort_Capability(def
                              LEQV(x,y: Entry): B)
            for Queue_Template;
      requires Is_Total_Preordering(LEQV);

    Definition IsNondecreasing(a: Str(Entry)): B=
      (for all b,c: Str(Entry),
          for all x,y: Entry,
            if a = b o <x> o <y> o c
              then LEQV(x,y));

    Operation Sort_Q(updates Q: Queue);
       ensures IsNondecreasing (Q)
                 and IsPermutation(#Q,Q);
end Sort_Capability;
```

2.1 Experimentation Part 1: Human Errors

The lessons learned in the first phase of this case study are likely typical of any such effort. They involved human errors in writing specifications and loop invariants. The implementation of the sort operation chosen for this verification experiment is a selection sort. The Sort_Q code (procedure) uses a local operation Remove_Min, only the specification of which is given. Its code also makes use of a programming operation to compare two entries; the operation has the behavior of the relation LEQV used in the specification. To use the sorting capability, a client would have to supply a compatible mathematical relation and an operation for ordering.

The procedure (code) for sorting defines and uses a local operation Remove_Min to find and remove a minimum element of a queue according to the ordering as shown in in Figure 2. The loop in the Sort_Q procedure is annotated by a software engineer with a list of variables that are modified by the loop, an invariant (expressed as a **maintaining** assertion), and a progress metric (**decreasing** expression) to establish total correctness.

Following the loop, :=: denotes the swap operator which can be used to exchange the values of two objects cleanly, without introducing aliasing [6].

We used our VC generator to generate verification conditions that represent the correctness of both the Remove_Min and Sort_Q procedures and attempted to prove them using Isabelle. The generated verification conditions correspond to the postcondition of the operation that is verified, precondi-

```
    Definition IsASmallest(a: Str(Entry);
                           x: Entry): B=
          (for all b,c: Str(Entry),
            for all y: Entry,
              if a = b o <y> o c then LEQV(x,y));

    Operation Remove_Min(updates Q: Queue;
                         replaces min: Entry);
       requires |Q| /= 0;
       ensures IsPermutation(Q o <min>, #Q)
                 and IsASmallest(Q, min);
-- code omitted

     Procedure Sort_Q(updates Q: Queue);
      Var sorted:Queue;
      Var min:Entry;
      While (Length(Q) /= 0)
       changing Q, sorted, min;
       maintaining IsPermutation(Q o sorted, #Q)
                   and IsNondecreasing(sorted);
       decreasing |Q|;
      do
         Remove_Min(Q, min);
         Enqueue(min, sorted);
      end;
      Q :=: sorted;

     end Sort_Q;
```

Figure 2: Original Sort_Q Implementation

```
    Operation Remove_Min(updates Q: Queue;
                         replaces min: Entry);
       requires |Q| /= 0;
       ensures IsPermutation(Q o <min>, #Q)
                 and IsASmallest(#Q, min);
-- code omitted

     Procedure Sort_Q(updates Q: Queue);
      Var sorted:Queue;
      Var min:Entry;
      While (Length(Q) /= 0)
       changing Q, sorted, min;
       maintaining IsPermutation(Q o sorted, #Q)
          and IsNondecreasing(sorted)
          and
           (for all y: Entry,
              if IsSubstring(<y>, Q) then
                   IsNondecreasing(sorted o <y>));
       decreasing |Q|;
       do
         Remove_Min(Q, min);
         Enqueue(min, sorted);
       end;
       Q :=: sorted;

     end Sort_Q;
```

Figure 3: Corrected Sort_Q Implementation

tions of operations that are called, and the correctness of the programmer-supplied loop invariant and progress metric.

Some of the verification conditions could not be proved. One them was the VC that represents the postcondition of Remove_Min. This VC could not be proved because of an incorrect specification of Remove_Min, which should have been:

```
ensures ... and IsASmallest(#Q, min);
```

Once the specification error was fixed, there was still a problem in proving the Sort_Q procedure. This was quickly traced to the loop invariant being too weak. While the two conjuncts in the loop invariant seem to match up properly with the two conjuncts in the postcondition of Sort_Q, the invariant IsNondecreasing(sorted) itself could not be established after min was enqueued. In particular, the invariant that every element in sorted is related to every element in Q was missing. This information is captured in the corrected code as shown in Figure 3.

2.2 Experimentation Part 2: Automation Difficulties

Once we corrected the annotations and generated new verification conditions, it was easy to check manually that each VC could be proved. In automating the proof with Isabelle, though a majority of the VCs could be proved, proofs of some VCs required human assistance.

The use of Isabelle as a tool to prove the VCs automatically requires the use of needed theories and lemmas from the RESOLVE mathematical theory library. Our approach is to write each RESOLVE mathematical theory (in this instance, string theory) as a new Isabelle theory, complete with defined functions and needed algebraic lemmas and theorems. We also leverage the already developed List theory in Isabelle for the internal representation of Strings (according to the recommended practice for Isabelle [15]); however, all VCs are proved with lemmas from String theory. Since we are focused on the ease and provability of VCs, in the present paper, we do not discuss the proofs of the lemmas themselves.

Once the basic theory is imported into Isabelle, lemmas about strings must be identified and tagged. The tags indicate to Isabelle how the lemma should be used in an automatic proof. Isabelle can use lemmas as simplification rules, introduction rules, destruction rules, and elimination rules. The simplification rules rewrite a term as another term. The introduction rules replace a goal of a lemma to be proved with the assumptions of an already proved lemma. The destruction rules replace a set of assumptions in a lemma to be proved with the goal of an already proved lemma. Essentially, how the lemmas are tagged determines how Isabelle will attempt to prove a VC automatically.

The VCs in this case are provable automatically, once the appropriate lemmas are tagged for use in the proofs, except for two cases discussed in some detail here. The first case is the proof of the loop invariant at the end of the while loop in the Remove_Min procedure as shown in Fig. 4. Here, the main issue is the development of the lemmas and theorems for the IsPermutation predicate. Once the IsPermutation predicate is expanded and the quantifiers instantiated appropriately, Isabelle can then prove the VC. A more well developed theory, with lemmas that allow for more powerful manipulations of the commutativity of concatenation within IsPermutation would help mitigate this issue.

The second case is shown in Figure 5. This lemma is the VC resulting from the obligation to prove the loop invariant at the end of the while loop in the Sort_Q procedure. In Figure 5, we use lemma ND4; the lemma is displayed in Figure 6. Lemma ND4 allows us to look at only the last entry in a string to determine whether adding another element would preserve the IsNondecreasing property. The main problem with this lemma is that Isabelle cannot find the proof automatically without being told that ND4 should be used and that is should be used in a particular manner, as an introduction rule. Without the knowledge that the usage of the lemma ND4 is always good, Isabelle stops exploring before a successful proof can be found. Once a person adds a manual declaration to apply the correct lemma (as shown

```
lemma 4:
"[|
 (min_int <= 0);
 (0 < max_int) ;
 (Max_Length > 0);
 is_initial min;
 (length Q <= Max_Length);
 length Q ~= 0;
 Q = (<min2> o Q3);
 IsPermutation ((temp1 o Q2) o <min1>) Q;
 IsASmallest (temp1 o <min1>) min1;
 (length Q2 ~= 0);
 Q2 = (<x1> o Q1);
 x1 <= min1
|]
==>
 IsPermutation (((temp1 o <min1>)
                 o Q1) o <x1>) Q "
apply auto
apply (unfold IsPermutation_def)
apply (auto)
apply (drule_tac x=x in spec)
apply auto
done
```

Figure 4: First Case of Problem VCs

by the "apply (rule ND4)" command in Figure 5), Isabelle's reasoner can find the proof.

```
lemma 10:
"[|
 (min_int <= 0);
 (0 < max_int);
 (Max_Length > 0);
 (length Q <= Max_Length);
 (IsPermutation (Q2 o sorted1) Q);
 (IsNondecreasing sorted1);
 ALL y. Is_SubString <y> Q2
        --> (IsNondecreasing (sorted1 o <y>));
 (length Q2 ~= 0);
 (IsPermutation (Q1 o <min1>) Q2);
 (IsASmallest Q2 min1)
|]
==>
 ALL y. Is_SubString <y> Q1 -->
    (IsNondecreasing ((sorted1 o <min1>) o <y>))"
apply auto
apply (rule ND4)
apply auto
done
```

Figure 5: Second Case of Problem VCs

The two issues raised in this subsection seem to be critical to the verification problem; namely the theory is not well developed enough (relative to the capabilities of the tool used) to permit the proof of VCs or the theory is well developed enough but the tool cannot construct the proof without hints such as the lemmas necessary or even when each lemma should be applied in a proof.

2.3 Experimentation Part 3: Impact of Theory Development on Automation

One of the difficulties in automation had to do with universal instantiation. Though this problem has received attention, our next line of thinking was to avoid creating this problem in the first place for automated provers. Our solution approach is to eliminate universal quantification in our annotations through new definitions, whenever possible. For our case study, this approach has led to a new definition, as well as development of new theorems involving the new definition. If we are successful in automating this totally, then it

```
lemma ND4:
"
[|
 IsNondecreasing (a o <y>);
 y <= b
|]
==>
  IsNondecreasing ((a o <y>) o <b>)
"
```

Figure 6: Lemma ND4

would also suggest a more general attempt to eliminate universal quantification through suitable mathematical theory development. Our verification system already allows for the use of ghost variables and manipulation of ghost variables to eliminate existential quantifiers in assertions (following King's principles [9] for the idea). This finding led us to the specification of Remove_Min operation and the invariant for the Sort_Q procedure as shown in Figures 7 and 8.

```
Definition IsPreceding ...

Operation Remove_Min(updates Q: Queue;
                     replaces min: Entry);
   requires |Q| /= 0;
   ensures IsPermutation(Q o <min>, #Q)
           and IsPreceding(<min>, #Q);
-- code omitted

 Procedure Sort_Q(updates Q: Queue);
  Var sorted:Queue;
  Var min:Entry;
  While (Length(Q) /= 0)
   changing Q, sorted, min;
   maintaining IsPermutation(Q o sorted, #Q)
        and IsNondecreasing(sorted)
        and IsPreceding(<min>, Q);
   decreasing |Q|;
   do
    Remove_Min(Q, min);
    Enqueue(min, sorted);
   end;
   Q :=: sorted;

 end Sort_Q;
```

Figure 7: Modified Theory Development for Specification

The new predicates are created and additional lemmas are proved that show how to the new predicates relate to others. The goal for this theory development is to eventually create enough new functions/predicates and relevant facts about those functions/predicates to remove the universal quantifiers from the specifications as we gain more experience with the proofs of the VCs. We envision large theory developments that are created *a priori* to be used in the specifications of programmatic procedures. These theory developments would be mostly fixed, e.g. most programming specifications would fall within such theories. Of course, when new theory development is needed, the newly created theories/functions/predicates would be added to the theory repository.

In this example, notice that the definitions of IsNondecreasing and IsPreceding are not usable elsewhere because they are specialized based on the user-supplied relation for ordering LEQV. A consequence of this deficiency is that any theorems that are proved for these definitions will also be specialized. Since the underlying principles are more general, it has led us to the following more gen-

```
Enhancement Sort_Capability(def LEQV(x,y: Entry): B)
            for Queue_Template;
      requires Is_Total_Preordering(LEQV);

    Operation Sort_Q(updates Q: Queue);
        ensures Is_Conformal_w (LEQV, Q)
                and IsPermutation(#Q,Q);
end Sort_Capability;
```

Figure 8: Modified Specification for Sort_Q Procedure

eral (and simple) specification of sorting. The predicate Is_Conformal_w takes an arbitrary binary relation and a string; it is a more general version of IsNondecreasing. The predicate Universally_Relates_to takes an arbitrary binary relation and two strings. It is a more general version of IsPreceding.

```
Begin String_Theory

...

Lemma PermutationCommutative
  IsPermutation (x o y) (y o x)
     ...

Definition Is_Conformal_w (R: (x: G, y: G) : B,
                           a : Str(G)) : B
  = for all x, y.
        If Is_Substring(<x> o <y>, a)
         Then R(x,y)

...

Definition Universally_Relates_to(
                           R: (x: G, y: G) :B,
                           a : Str(G),
                           b : Str(G))i : B
  = for all x, y. If Is_Substring(<x>, a)
                  and Is_Substring(<y>, b)
                Then R(x,y)

...

...
End String_Theory
```

Figure 9: Definitions of Is_Conformal_w and Universally_Relates_to

The definitions of these predicates are shown in Figure 9. Given the theory development for these general definitions, we do not foresee any additional automation problems due to the generalization.

3. DISCUSSION

An excellent summary of several verification efforts may be found in [12], and details of efforts more directly related to the present paper may be found in [7]. Here, we present only a summary of most related efforts for full, automated verification of programs.

To our knowledge there is no system available today for automated and modular verification of full behavior of object-based programs. Using the πVC Verification System [1], Bradley, Manna, and Sipma have verified several sorting algorithms (of integer arrays) written in the pi programming language. A few examples of the sorting algorithms proved in πVC are: insertion sort, merge sort, bubble sort, and quick sort for integer arrays in the pi programming language. The verification of the various sorting algorithms that they implemented each required less than 20 seconds to complete. Unlike our work, their focus is on using specialized data types and decision procedures.

The Why software verification system [5] can be used to generate verification conditions, similar to the ones generated by our tool. Unlike our effort, the Why tool focuses primarily on functional programs annotated with assertions and typically relies on built-in types (ie, arrays).

Verisoft [13] is a verification system that relies on Isabelle to perform the actual proofs, as we have done. It is based on a "clean" subset of the C programming language, C0 and it aims for full verification. Since it is focused on C, it does not address modular verification of programs using objects.

4. CONCLUSION

We have presented our experiences with the verification of a sorting algorithm implementation using the proof assistant Isabelle. The lessons learned from this exercise include information about the types of errors programmers are likely to make in writing the annotated code, and some of the possible trouble areas for a proof assistant such as Isabelle. We plan to continue this work in order to address the identified issues with Isabelle. At least as importantly, we have also learned how automation can lead to better software engineering and sophisticated theory development along dimensions that would not have come into focus without attempting automated verification in the first place. Finally, the case study illustrates at least a set of minimum requirements for an integrated language for developing verified software.

Acknowledgments

We would like to thank Jeremy Avigad for his guidance in connecting our verification effort with the Isabelle prover and Bill Ogden for his help in formulating the assertions used in this paper. Our thanks are also due to Harvey Friedman and Bruce Weide for various discussions relating to the topics of this paper. We would also like to thank Bruce Adcock, Derek Bronish, Wayne Heym, Joan Krone, Tim Long, Brandon Mintern, and Anna Wolf for their suggestions.

This work was supported in part by the National Science Foundation under grants DMS-0701187, DMS-0700174, DMS-0701260 and DUE-0633055.

5. REFERENCES

[1] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[2] P. Bucci. Osu rsrg component catalog, 2008. `http://www.cse.ohio-state.edu/rsrg/sce/rcpp/RESOLVE_Catalog-HTML/index.%html`.

[3] P. Bucci, J. E. Hollingsworth, J. Krone, and B. W. Weide. Part iii: implementing components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):40–51, 1994.

[4] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part ii: specifying components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590/2007 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer-Verlag.

[6] D. Harms and B. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[7] H. Harton, J. Krone, and M. Sitaraman. Formal program verification. *Encyc. of Computer Science and Engineering*, 2008 (to appear).

[8] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[10] G. Kulczycki. *Direct Reasoning*. PhD thesis, Clemson University, 2004.

[11] G. Kulczycki, M. Sitaraman, N. Yasmin, and K. Roche. Formal specification. *Encyc. of Computer Science and Engineering*, 2008 (to appear).

[12] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006. ACM.

[13] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. *sefm*, 0:2–12, 2005.

[14] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part i: the resolve framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes*, 19(4):23–28, 1994.

[17] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[18] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.

[19] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Fraiser. Incremental benchmarks for software verification tools and techniques. Technical Report RSRG-08-02, School of Computing, Clemson University, 2008.

[20] C. Weidenbach. Spass - version 0.49. *J. Autom. Reason.*, 18(2):247–252, 1997.

[21] J. Woodcock and R. Banach. The verification grand challenge. *Journal of Universal Computer Science*, 13(5):661–668, may 2007. `http://www.jucs.org/jucs_13_5/the_verification_grand_challenge`.