Rajeev Joshi
Peter Müller
Andreas Podelski (Eds.)

LNCS 7152

Verified Software: Theories, Tools, Experiments

4th International Conference, VSTTE 2012
Philadelphia, PA, USA, January 2012
Proceedings

Springer

Rajeev Joshi   Peter Müller
Andreas Podelski (Eds.)

Verified Software: Theories, Tools, Experiments

4th International Conference, VSTTE 2012
Philadelphia, PA, USA, January 28-29, 2012
Proceedings

Springer

Volume Editors

Rajeev Joshi
MS 301-285
4800 Oak Grove Drive, Pasadena, CA 91109, USA
E-mail: rajeev.joshi@jpl.nasa.gov

Peter Müller
ETH Zürich
Universitätstr. 6, 8092 Zürich, Switzerland
E-mail: peter.mueller@inf.ethz.ch

Andreas Podelski
University of Freiburg
Department of Computer Science
Georges-Köhler-Allee 52, 79110 Freiburg, Germany
E-mail: podelski@informatik.uni-freiburg.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-27704-7 e-ISBN 978-3-642-27705-4
DOI 10.1007/978-3-642-27705-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011944848

CR Subject Classification (1998): D.2, F.3, D.3, D.1, C.2, F.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at the 4th International Conference on Verified Software: Theories, Tool and Experiments (VSTTE), which was held in Philadelphia, USA, during January 28–29, 2012. Historically, the conference originated from the Verified Software Initiative (VSI), a cooperative, international initiative directed at the scientific challenges of large-scale software verification. The inaugral VSTTE conference was held at ETH Zurich in October 2005. Starting in 2008, the conference became a biennial event: VSTTE 2008 was held in Toronto, and VSTTE 2010 was held in Edinburgh.

The goal of the VSTTE conference is to advance the state of the art through the interaction of theory development, tool evolution, and experimental validation. Topics of interest include:

- Specification and verification techniques
- Tool support for specification languages
- Tool for various design methodologies
- Tool integration and plug-ins
- Automation in formal verification
- Tool comparisons and benchmark repositories
- Combination of tools and techniques (e.g., formal vs. semiformal, software specification vs. engineering techniques)
- Customizing tools for particular applications
- Challenge problems
- Refinement methodologies
- Requirements modeling
- Specification languages
- Specification/verification case studies
- Software design methods
- Program logic

The conference received 54 submissions, of which 20 were accepted after a rigorous review process, for an acceptance rate of 37%.

The conference included two invited talks, by Rupak Majumdar (Max Planck Institute for Software Systems) and Wolfgang Paul (Saarland University), and two tutorials, by Rustan Leino (Microsoft Research) and Francesco Logozzo (Microsoft Research).

A software verification competition was also held in advance of the main conference. This competition consisted of five independent programs, which had to be verified using automated verification tools. The competition was held online, November 8–10, 2011, and was a great success, with 29 participating teams, comprising 79 individuals, and 22 verification tools. Competition results were announced on December 5, 2011.

We would like to thank the invited speakers, all submitting authors, the organizers of the verification competition, the Steering Committee, the General Chair, the external reviewers, and especially the Program Committee, who put in a lot of hard work into reviewing and selecting the papers that appear in this volume.

November 2011

Rajeev Joshi
Peter Müller
Andreas Podelski

Organization

Program Committee

Clark Barrett	New York University, USA
Lars Birkedal	IT University of Copenhagen, Denmark
Patrick Cousot	Courant Institute - New York University and École normale supérieure, USA and France
Leonardo De Moura	Microsoft Research, USA
Jean-Christophe Filliatre	CNRS, France
John Hatcliff	Kansas State University, USA
Bart Jacobs	Katholieke Universiteit Leuven, Belgium
Ranjit Jhala	UC San Diego, USA
Rajeev Joshi	Laboratory for Reliable Software, Jet Propulsion Laboratory, USA
Gerwin Klein	NICTA and UNSW, Australia
Viktor Kuncak	EPFL, Switzerland
Gary T. Leavens	University of Central Florida, USA
Rustan Leino	Microsoft Research, USA
Panagiotis Manolios	Northeastern University, USA
Peter Müller	ETH Zurich, Switzerland
Tobias Nipkow	TU Munich, Germany
Matthew Parkinson	Micrsosoft Research, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Wolfgang Paul	Saarland University, Germany
Andreas Podelski	University of Freiburg, Germany
Natasha Sharygina	Università della Svizzera Italiana, Switzerland
Willem Visser	Stellenbosch University, South Africa
Thomas Wies	IST Austria

Additional Reviewers

Alberti, Francesco
Andronick, June
Apel, Sven
Bagherzadeh, Mehdi
Bengtson, Jesper
Blanchet, Bruno
Boyton, Andrew
Brim, Lubos
Butterfield, Andrew
Chamarthi, Harsh Raju
Chlipala, Adam
Daum, Matthias
Dinsdale-Young, Thomas
Dross, Claire
Fedyukovich, Grigory
Feret, Jérôme
Goldberg, Eugene
Greenaway, David
Gvero, Tihomir
Hadarean, Liana

Haddad, Ghaith
Hansen, Michael R.
Hildebrandt, Thomas
Hobor, Aquinas
Hojjat, Hossein
Hussain, Faraz
Jacobs, Swen
Jain, Mitesh
Jeong, Mina
Jovanovic, Dejan
King, Tim
Kolanski, Rafal
Lahiri, Shuvendu
Laviron, Vincent
Losa, Giuliano
Martel, Matthieu
Massé, Damien
Mauborgne, Laurent
Mery, Dominique
Milicevic, Aleksandar
Miné, Antoine
Murray, Toby
Neis, Georg
Norrish, Michael
Ouaknine, Joel
Owens, Scott
Papavasileiou, Vasilis
Paskevich, Andrei
Pichardie, David
Rollini, Simone Fulvio
Rozier, Kristin Yvonne
Sery, Ondrej
Smans, Jan
Suter, Philippe
Svendsen, Kasper
Tkachuk, Oksana
Tsitovich, Aliaksei
Vogels, Frédéric
Vujosevic-Janicic, Milena
Wang, Wei
Wickerson, John
Zufferey, Damien

Table of Contents

Cyber War, Formal Verification and Certified Infrastructure

Wolfgang Paul

Saarland University, Germany
wjp@cs.uni-saarland.de

Abstract. Cyber war is recognized to be real. Like all wars this is bad for many people, but not for manufacturers of weapons. An attacker in cyber war can be defined as an unauthorized user process without access to physical side channels; with physical access we would be back to ordinary warfare and espionage. IT infrastructure which - by separation theorems - can be guaranteed to be immune against such attackers is therefore a defensive weapon in cyber war. The verification community is the only potential manufacturer of such infrastructure and thus has a chance to access resources vastly superior to those for ordinary research and development.

In order to develop such infrastructure, one would have to

1. develop a pervasive mathematical theory of IT infrastructure
2. formally verify it
3. convince industry and politicians, that the specifications of this theory are meaningful and
4. convince certification agencies, that all verification tools involved are sound

Problem (3) could be solved by providing standardized machine readable specifications of the usual components of IT infrastructure. Note that agreeing on standards is usually a non trivial sociological process. (1), (2) and (4) are 'ordinary' technical problems. In the main part of this talk we will review the state of the art for these problems and estimate the resources to resolve the remaining open subproblems. The resulting costs are large compared to normal research budgets and very small compared to the cost of war.

A Certified Multi-prover Verification Condition Generator*

Paolo Herms[1,2,3], Claude Marché[2,3], and Benjamin Monate[1]

[1] CEA, LIST, Lab. de Sûreté du Logiciel, Gif-sur-Yvette F-91191
[2] INRIA Saclay - Île-de-France, 4 rue Jacques Monod, Orsay, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract. Deduction-based software verification tools have reached a maturity allowing them to be used in industrial context where a very high level of assurance is required. This raises the question of the level of confidence we can grant to the tools themselves. We present a certified implementation of a verification condition generator. An originality is its genericity with respect to the logical context, which allows us to produce proof obligations for a large class of theorem provers.

1 Introduction

Among the various classes of approaches to static verification of programs, the so-called *deductive verification* approach amounts to verifying that a program satisfies a given behavioral specification by means of theorem proving. Typically, given a program and a formal specification, a verification condition generator produces a set of logical formulas, that must be shown to be valid by some theorem prover. Deductive verification tools have nowadays reached a maturity allowing them to be used in industrial context where a very high level of assurance is required [25]. This raises the question of the level of confidence we can grant to the tools themselves. This is the question we address in this paper.

One can distinguish two main kinds of deductive verification approaches. The first kind is characterized by the use of a deep embedding of the input programming language in a general purpose proof assistant. One of the earlier work of this kind is done in the SunRise system in 1995 [17] where a simple imperative language is defined in HOL, with a formal operational semantics. A set of Hoare-style deduction rules are then shown valid. A SunRise program can then be specified using HOL assertions, and proved in the HOL environment.

The second kind of approaches provide standalone verification condition generators automatically producing verification conditions, usually by means of variants of Dijkstra's weakest precondition calculus. This is the case of ESC/Java [10], B [1] ; the Why platform [14] and its Java [22] and C [13] front-ends ; and Spec# [3] and VCC [11] which are front-ends to Boogie [2]. Being independent of

* This work is partly supported by the *U3CAT* project (ANR-08-SEGI-021, http://frama-c.com/u3cat/) of the French national research organization (ANR).

any underlying proof assistant, these tools analyze programs where formal specifications are given in ad-hoc annotation language such as JML [7] and ACSL [4]. However, up to now these standalone tools have never been formally proved to be sound.

Our goal is to combine the best of both approaches: a guaranteed sound VC generator, able to produce VCs for multiple provers. We implement and prove sound, in the Coq proof assistant [5], a VC generator inspired by the former Why tool. To make it usable with arbitrary theorem provers as back-ends, we make it generic with respect to a *logical context*, containing arbitrary abstract data types and axiomatizations. Such a generic aspect is suitable to formalize memory models needed to design front-ends for mainstream programming language, as it is done for C by VCC above Boogie or Frama-C/Jessie above Why. The input programs of our VC generator are imperative programs written in a core language which operates on mutable variables whose values are data types of the logical context. The output logic formulas are built upon the same logical context. This certified Coq implementation is crafted so it can be extracted into a standalone executable.

Section 2 formalizes our notion of generic logical contexts. Section 3 formalizes our core language, and defines its operational semantics. Section 4 defines the weakest precondition computation WP and proves its *soundness*. Theorem 1 states that if for each function of a program, its pre-condition implies the WP of its post-condition, then all its annotations are satisfied. Section 5 aims at the extraction of a standalone executable. We introduce a variant wp of the calculus which produces concrete formulas instead of Coq ones. Theorem 2 states that wp obligations imply the WP obligations. The main result is then Theorem 3 which states the soundness of the complete VC generation process. We conclude in Section 6 by comparing with related works and discussing perspectives. Due to lack of space, we sometimes refer to our technical report [16] for technical details. The sources of the underlying Coq development are available at `http://www.lri.fr/~herms/` .

2 Logical Contexts

Our background logic is multi-sorted first-order logic with equality. Models for specifying programs can be defined by declaring types, constant, function and predicate symbols and axioms. Models may integrate predefined theories, a typical example being integer arithmetic.

Definition 1. *A* logical signature *is composed of (1) a set* utype *of sort names introduced by the user ; (2) a set* sym *of constant, function and predicate symbols; (3) a set* ref *of global reference names and (4) a set* exc *of exceptions names. The set of all data types is defined by the grammar*

$$\textit{type} ::= \mathrm{Tuser}\ \textit{utype} \mid \mathrm{Tarrow}\ \textit{type}\ \textit{type} \mid \mathrm{Tprop}$$

that is, the types are completed with built-in types for propositions and functions. We require every symbol, exception and reference to have an associated

```
type array
logic select : array -> int -> int
logic store :
 array -> int -> int -> array
axiom select_eq: forall a,i,x.
 select (store a i x) i = x
axiom select_neq : forall a,i,j,x.
 i <> j ->
  select (store a i x) j =
  select a j

logic sorted : array -> int -> prop
axiom sorted_def: forall a,n.
 sorted a n <->
 forall i,j. 0 <= i <= j < n ->
  select a i <= select a j

logic swap : array -> array ->
              int -> int -> prop
```

```
axiom swap_def: forall a,b,i,j.
 swap a b i j <->
 select a i = select b j /\
 select a j = select b i /\
 forall k. k <> i /\ k <> j ->
  select a k = select b k

logic permut:
 array -> array -> int -> prop
axiom permut_refl: forall a,n.
 permut a a n
axiom permut_sym: forall a,b,n.
 permut a b n -> permut b a n
axiom permut_trans:
 forall a,b,c,n.
  permut a b n /\ permut b c n ->
  permut a c n
axiom permut_swap:
 forall a,b,i,j,n.
  0 <= i < n /\ 0 <= j < n /\
  swap a b i j -> permut a b n
```

Fig. 1. Logical context for sorting

type. In our Coq implementation, *sym*, *ref*, and *exc* are of type $type \rightarrow$ Type (see the report for details [16]). The parameter of the latter are written as subscript in the following.

Example 1. Fig. 1 presents an appropriate model for specifying a program for sorting an array. An abstract type `array` is introduced to model arrays of integers indexed by integers. It is axiomatized with the well-known theory of arrays. We also define predicates (`sorted` t i) meaning that $t[0], \ldots, t[i-1]$ is an increasing sequence, and (`permut` t_1 t_2) meaning that t_1 is a permutation of t_2. The latter is axiomatized: it is an equivalence relation that contains all transpositions (`swap`) of two elements.

The logical signature of this example is thus given by $utype = \{\mathtt{array}\}$ and $sym = \{\mathtt{select}, \mathtt{store}, \mathtt{sorted}, \mathtt{swap}, \mathtt{permut}\}$ (*ref* and *exc* will come later). Each symbol is annotated by the appropriate *type*, e.g. $\mathtt{select} : sym_{(\mathrm{Tarrow}(\mathrm{Tuser}\ \mathtt{array})(\mathrm{Tarrow}\ \mathrm{Tint}\ \mathrm{Tint}))}$.

2.1 Dependently Typed *de Bruijn* Indices

A design choice in our formalization is to define terms and expressions such that they are well typed by construction. This simplifies the definition of the semantics and the weakest precondition calculus on such expressions, as we don't need to handle malformed constructions at those points. To begin we need to ensure that occurrences of variables actually correspond to bound variables in their current scopes and that they are used with the correct type. Here we use so-called dependently typed *de Bruijn* indices following the preliminary approach of Herms [15] as documented in [8].

Dependent indices are like regular *de Bruijn* indices, in that I_0 refers to the innermost bound variable, $(I_S\ I_0)$ to the second innermost bound variable, etc. Additionally they carry information about their typing environment and about the type of the variable they represent. We use indices of type $idx_{A,E}$ to represent variables of type A under a typing environment E, that is the list of the types of the bound variables. The type of the innermost bound variable is stored at the first position in the typing environment, the type of the second innermost bound variable at the second position, etc. In Coq we can formalize this constraint about the valid parameters of *idx* by assigning its constructors the types $I_0 : idx_{A,A::E}$ and $I_S : idx_{A,E} \rightarrow idx_{A,B::E}$ (see [16]).

Dependent indices are thus placeholders within terms but they can also be used to reference elements within heterogeneous lists. In such a heterogeneous list each element may have a different type. The type $hlist_E$ of heterogeneous lists then depends on the list of types E of their elements. Thanks to the constraints on the type parameters, if an index $i : idx_{A,E}$ references an element within a heterogeneous list $l : hlist_E$, we are sure to find an element of type A at i-th position of l. This allows us to define the function $accsidx : idx_{A,E} \rightarrow hlist_E \rightarrow A$ which recursively accesses elements within a heterogeneous list.

We will use these heterogeneous lists to give semantics to our languages. Precisely, heterogeneous lists are the representation of evaluation environments which associate a value to each variable in the current typing environment. The function *accsidx* is then used in the semantics rule for variable access.

Example 2. The heterogeneous list $l = [5; true; succ]$ has type $hlist\ [\mathbb{Z}; bool; \mathbb{Z} \rightarrow \mathbb{Z}]$. *De Bruijn* indices $I_0 : idx_{\mathbb{Z},[\mathbb{Z};bool;\mathbb{Z}\rightarrow\mathbb{Z}]}$ and $I_S\ (I_S\ I_0) : idx_{(\mathbb{Z}\rightarrow\mathbb{Z}),[\mathbb{Z};bool;\mathbb{Z}\rightarrow\mathbb{Z}]}$, are well-typed and can be used to access their values, e.g. $accsidx\ I_0\ l = 5 : \mathbb{Z}$ and $accsidx\ (I_S\ I_0)\ l = true : bool$.

2.2 Terms and Propositions

Terms and propositions follow the usual classical first-order logic. For the need of programs, we add the declaration of local names using `let` binders, the access to a reference r (with concrete syntax `!r`) and the dereferencing of such a reference in a former state labeled by l (concrete syntax `r@l`). Labels are represented by bounded integers and new labels can be declared at the expression level.

The formal syntax of terms and propositions is given in Fig. 2. Terms $t_{L,E,A}$ and propositions $p_{L,E}$ depend on the parameters E and L, denoting respectively the typing environment and the highest index of a valid label. Terms additionally depend on the parameter A, the type of the value they denote. Variables are represented by our dependent indices $idx_{A,E}$. The constructor Tlet expresses let-blocks at the term level. As usual with *de Bruijn* indices, no variable name is given and the body of the block is typed in a typing environment that is enriched

$$\begin{array}{rcll} t_{L,E,A} & ::= & \text{Tconst } \mathit{sym}_A & \\ & | & \text{Tvar } \mathit{idx}_{A,E} & \\ & | & \text{Tapp } t_{L,E,(\text{Tarrow}BA)} \; t_{L,E,B} & \\ & | & \text{Tlet } t_{L,E,B} \; t_{L,B::E,A} & \\ & | & \text{Tderef } \mathit{ref}_A & (\text{* } \mathtt{!r} \text{ *}) \\ & | & \text{Tat } \mathit{label}_L \; \mathit{ref}_A & (\text{* } \mathtt{r@l} \text{ *}) \end{array} \qquad \begin{array}{rcl} p_{L,E} & ::= & \text{Peq } t_{L,E,A} \; t_{L,E,A} \\ & | & \text{Pand } p_{L,E} \; p_{L,E} \\ & | & \text{Pimply } p_{L,E} \; p_{L,E} \\ & | & \text{Pforall } p_{L,A::E} \\ & | & \text{Plet } t_{L,E,A} \; p_{L,A::E} \\ & | & \text{Pfalse} \\ & | & \text{Pterm } t_{L,E,\text{Tprop}} \end{array}$$

Fig. 2. Inductive definitions of terms and propositions

by the type of the term to be remembered. The symbol application is formalized in a curryfied style. For the propositions we define only the ones needed within the WP calculus. The constructor Pterm allows to construct user-defined atomic propositions from terms. As Tlet at the term-level, Plet expresses let-blocks at the level of props and binds a new *de Bruijn* variable. Similarly Pforall binds a new de Bruijn variable but generalizing it instead of assigning a value to it. The Pforall and the Plet bind a new *de Bruijn* variable.

2.3 Logical Contexts, Semantics

The semantics of our generic language depends on an *interpretation* given to types and symbols. From such an interpretation, any term or proposition can be given a value, in a given *environment* for variables and given *state* for references.

Given a logical signature, an *interpretation* is a pair of a function *denutype* giving an interpretation of the user types, and a function *densym* giving an interpretation of the introduced function and predicate symbols. Given *denutype* we define *dentype* to interpret all types. An *evaluation environment* Γ of type env_E is a heterogeneous list as described above. A *memory state* S of type state_L is a vector of size L of mappings from references ref_A to values of type $\mathit{dentype} A$. The first element denotes the current state whereas the $(l+1)$-nth element denotes the state labeled by l. This is the reason for the L-parameter of terms and propositions. A term of type $t_{L,E}$ can be safely evaluated in a state of type state_L. As a special case, a state_0 is only composed of the current state and $t_{0,E}$ cannot contain any labeled dereferenciation at all. The semantics of terms is defined by structural recursion (Fig. 3), where we use the syntactic sugar $\mathit{Here}\ S = S[0]$ and $\mathit{At}\ S\ l = S[l+1]$ by analogy to the syntax. Note how the rules for Tlet and Pforall push the newly bound variable into Γ. Note also how correct typing is ensured by construction.

A *logical context* is a pair of a logical signature and a set of axioms over it. The programs that will be written in the next section will assume a given logical context. The goal is to prove them valid with respect to *any* interpretation which makes the axioms of that context valid: this will allow us to use various provers to discharge them.

$$\begin{aligned}
[\![\text{Tconst } s]\!]_{\Gamma,S} &::= \mathit{densym}\ s \\
[\![\text{Tvar } v]\!]_{\Gamma,S} &::= \mathit{accsidx}\ \Gamma\ v \\
[\![\text{Tderef } r]\!]_{\Gamma,S} &::= \mathit{Here}\ S\ r \\
[\![\text{Tapp } t_1\ t_2]\!]_{\Gamma,S} &::= ([\![t_1]\!]_{\Gamma,S}\ [\![t_2]\!]_{\Gamma,S}) \\
[\![\text{Tlet } t_1\ t_2]\!]_{\Gamma,S} &::= [\![t_2]\!]_{[\![t_1]\!]_{\Gamma,S}::\Gamma,S} \\
[\![\text{Tat } l\ r]\!]_{\Gamma,S} &::= \mathit{At}\ S\ l\ r
\end{aligned}
\qquad
\begin{aligned}
[\![\text{Peq } t_1\ t_2]\!]_{\Gamma,S} &::= [\![t_1]\!]_{\Gamma,S} = [\![t_2]\!]_{\Gamma,S} \\
[\![\text{Pand } p_1\ p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \wedge [\![p_2]\!]_{\Gamma,S} \\
[\![\text{Pimply } p_1\ p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \rightarrow [\![p_2]\!]_{\Gamma,S} \\
[\![\text{Pforall } p]\!]_{\Gamma,S} &::= \forall b : B,\ [\![p]\!]_{b::\Gamma,S} \\
[\![\text{Plet } t\ p]\!]_{\Gamma,S} &::= [\![p]\!]_{[\![t]\!]_{\Gamma,S}::\Gamma,S} \\
[\![\text{Pfalse}]\!]_{\Gamma,S} &::= \bot \\
[\![\text{Pterm } t]\!]_{\Gamma,S} &::= [\![t]\!]_{\Gamma,S}
\end{aligned}$$

Fig. 3. Denotational semantics of terms and propositions

3 The Core Programming Language

3.1 Informal Description

Our core language follows most of the design choices of the input language of Why. Indeed we reduce to an even more basic set of constructs, nevertheless remaining expressive enough to encode higher-level sequential algorithms. We follow an ML-style syntax; in particular there is no distinction between expressions and instructions. A program in this language is defined by a logical context and a finite set of function definitions, denoted f below, which can modify the global references of the context and can be mutually recursive.

Following again the Why design, our core language contains an exception mechanism, providing powerful control flow structures. As we will see these can be handled by weakest pre-condition calculus without major difficulty. Loops are infinite ones, with a given invariant. The only way to exit them is by using exceptions. We use $e_1; e_2$ as a shortcut for `let` $v = e_1$ `in` e_2 when the variable v is unused.

A definition of a function follows the structure

$$\texttt{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \{\ p\ \}\ e\ \{\ q\ \}$$

where the predicates p and q are the pre- and the post-condition. The types are those declared in the logical context. In the post-condition, the reserved name *result* is locally bound and denotes the result of the function of type τ and label *Old* is bound to denote the pre-state. Note that exceptions are not supposed to escape function bodies. We could easily support such a feature by adding a family of post-conditions indexed by exceptions as in Why [12].

Example 3. In Fig. 4 is a program that sorts the global array t by the classical selection sort algorithm. Note the use of the exception `Break` to exit from the infinite loops. Note also the use of labels in annotations, allowing to specify assertions, loop invariants and post-conditions that link up various states of execution.

3.2 Formal Syntax of Expressions

Like terms of the logic, expressions of programs are formalized by an inductive type $e_{L,E,A}$ depending on the parameters A, E and L, denoting respectively

```
ref t : array

let swap(i:int, j:int) : unit =
 { true }
 let tmp = select !t i in
 t := store !t i (select !t j);
 t := store !t j tmp
 { swap !t t@Old i j }

ref mi, mv, i, j : int
exc Break : unit

let selection_sort(n:int) : unit =
 { n >= 1 }
 i := 0;
 try loop
  { invariant 0 <= !i < n /\
    sorted !t i /\
    permut !t t@Old n /\
    forall k1,k2.
     0 <= k1 < i <= k2 < n ->
      select !t k1 <= select !t k2 }
  if !i >= n-1
   then raise (Break ()) else ();
```

```
   (* look for minimum value
      among t[i..n-1] *)
   mv := select !t !i; mi := !i;
   j := !i+1;
   try loop
    { invariant !i < !j /\
       !i <= !mi < n /\
       !mv = select !t !mi /\
      forall k. !i <= k < !j ->
       select !t k >= !mv }
    if !j >= n
     then raise (Break ()) else ();
    if select !t !j < !mv
     then (mi := !j ;
           mv := select !t !j)
     else ();
    j := !j + 1
   catch Break(v) ();
  label Lab:
   swap(!i,!mi);
   assert { permut !t t@Lab n } in
   i := !i + 1
  catch Break(v) ();
 { sorted !t n /\ permut !t t@Old n }
```

Fig. 4. Selection sort in our core language

$e_{L,E,A}$ ::=	Eterm $t_{L,E,A}$	pure term t
	| Elet $e_{L,E,B}$ $e_{L,B::E,A}$	`let` $v = e_1$ `in` e_2
	| Eassign ref_A $t_{L,E,A}$	r `:=` t
	| Eassert $p_{L,E}$ $e_{L,E,A}$	`assert {` p `} in` e
	| Eraise exc_A $t_{L,E,A}$	`raise` $(ex\ t)$
	| Eif $p_{L,E}$ $e_{L,E,A}$ $e_{L,E,A}$	`if` p `then` e_1 `else` e_2
	| Eloop $p_{L,E}$ $e_{L,E,B}$	`loop { invariant` p `}` e
	| Etry $e_{L,E,A}$ exc_B $e_{L,B::E,A}$	`try` e_1 `catch` $ex(v)$ e_2
	| Elab $e_{L+1,E,A}$	`label` l`:` e
	| Ecall $f_{A,P}$ $(t_{L,E,P_1}, ..., t_{L,E,P_n})$	call to f

Fig. 5. Inductive definition of expressions

the evaluation type, the typing environment and the highest index of a valid label. Abstract syntax of expressions including comprehensive type annotations is given in Fig 5. Notice that variables v and label l are left implicit in the inductive definition thanks to *de Bruijn* representation. Additionally expressions depend on a parameter F meaning the list of *signatures* of the functions in the program the expression can appear in. A signature is a pair of the return type of the function and the list of the function's parameters. F appears within expressions in function calls where we use dependent indexes to refer to functions, $f_{A,P} := idx_{\langle A,P\rangle,F}$. A function identifier is therefore an index pointing to an

$$\frac{}{\Gamma, S, \text{Eterm } t \Rightarrow S, [\![t]\!]_{\Gamma,S}} \qquad \frac{\Gamma, S, e_1 \Rightarrow S', v \qquad v :: \Gamma, S', e_2 \Rightarrow S'', o}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow S'', o}$$

$$\frac{}{\Gamma, S, \text{Eassign } r\ t \Rightarrow S\left[r/[\![t]\!]_{\Gamma,S}\right], [\![t]\!]_{\Gamma,S}} \qquad \frac{\Gamma, S, e_1 \Rightarrow S', ex\,(v)}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow S', ex\,(v)}$$

$$\frac{\Gamma, S\!\uparrow, e \Rightarrow S', o}{\Gamma, S, \text{Elabel } e \Rightarrow S'\!\downarrow, o} \qquad \frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e \Rightarrow S', o}{\Gamma, S, \text{Eassert } p\ e \Rightarrow S', o}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_1 \Rightarrow S', o}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow S', o} \qquad \frac{\neg[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_2 \Rightarrow S', o}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow S', o}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow S', v \qquad S', \text{Eloop } p\ e \Rightarrow S'', o}{S, \text{Eloop } p\ e \Rightarrow S'', o}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow S', ex(v)}{S, \text{Eloop } p\ e \Rightarrow S', ex(v)} \qquad \frac{}{\Gamma, S, \text{Eraise } ex\ t \Rightarrow S, ex([\![t]\!]_{\Gamma,S})}$$

$$\frac{S, e_1 \Rightarrow S', o \qquad o \neq ex}{S, \text{Etry } e_1\ ex\ e_2 \Rightarrow S', o} \qquad \frac{S, e_1 \Rightarrow S', ex(v) \qquad v :: \Gamma, S', e_2 \Rightarrow S'', o}{S, \text{Etry } e_1\ ex\ e_2 \Rightarrow S'', o}$$

$$\frac{\Gamma_f := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}] \qquad [\![\mathsf{pre}_f]\!]_{\Gamma_f,S} \qquad \Gamma_f, S, \mathsf{body}_f \Rightarrow S', v \qquad [\![\mathsf{post}_f]\!]_{v::\Gamma_f,S'}}{\Gamma, S, \text{Ecall } f\ (t_1, ..., t_n) \Rightarrow S', v}$$

Fig. 6. Operational semantics of terminating expressions

element with the signature $\langle A, P\rangle$ within a heterogeneous list of types F. This heterogeneous list $hlist_{func\ F,F}$ is precisely the representation of a program pr_F, where each element is a function $func_{F,\langle A,P\rangle}$.

A function $func_{F,\langle A,P\rangle}$ consists of a body $e_{F,1,E,A}$, a pre-condition $p_{0,P}$ and a post-condition $p_{1,A::P}$. In the pre-condition no labels may appear, hence its type has the parameter 0. In the post-condition we allow referring to the pre-state of a function call: in the syntax this corresponds to using the label *Old*. The post-condition may additionally refer to the result of the function, hence its type environment is enriched by A. Note that in the definition of programs the parameter F appears twice: once as parameter of *hlist*, to define the signatures of the functions in the program, and once as parameter of *func* to constrain expressions in function bodies to refer only to functions with a signature appearing in F. This way we ensure the well-formedness of the graph structure of programs.

3.3 Operational Semantics

The operational semantics is defined in big-step style following the approach of Leroy and Grall [20]. A first set of inference rules inductively defines the semantics of terminating expressions (Fig. 6) and a second set defines the semantics of non-terminating expressions, co-inductively (Fig. 7). Judgement $\Gamma, S, e \Rightarrow S', o$ expresses that in environment Γ and state S, the execution of expression e terminates, in a state S' with *outcome* o: either a normal value v or an exception $ex(v)$ where v is the value held by it. There are two rules for `let` e_1 `in` e_2 depending on the outcome of e_1. The rule for assignment uses the update operation $S[r/a]$

$$\frac{\Gamma, S, e_1 \Rightarrow \infty}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow \infty} \qquad \frac{\Gamma, S, e_1 \Rightarrow S', v \qquad v :: \Gamma, S', e_2 \Rightarrow \infty}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_1 \Rightarrow \infty}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow \infty} \qquad \frac{\neg[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_2 \Rightarrow \infty}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow \infty}$$

$$\frac{\Gamma, S\!\uparrow, e \Rightarrow \infty}{\Gamma, S, \text{Elabel } e \Rightarrow \infty} \qquad \frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e \Rightarrow \infty}{\Gamma, S, \text{Eassert } p\ e \Rightarrow \infty} \qquad \frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow \infty}{S, \text{Eloop } p\ e \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow S', v \qquad S', \text{Eloop } p\ e \Rightarrow \infty}{S, \text{Eloop } p\ e \Rightarrow \infty}$$

$$\frac{S, e_1 \Rightarrow \infty}{S, \texttt{try}\, e_1\, \texttt{catch}\ ex()\ e_2 \Rightarrow \infty} \qquad \frac{S, e_1 \Rightarrow S', ex(v) \qquad v :: \Gamma, S', e_2 \Rightarrow \infty}{S, \text{try } e_1\ ex\ e_2 \Rightarrow \infty}$$

$$\frac{\Gamma_f := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}] \qquad [\![\mathsf{pre}_f]\!]_{\Gamma_f,S} \qquad \Gamma_f, S, \mathsf{body}_f \Rightarrow \infty}{\Gamma, S, \text{Ecall } f\ (t_1, ..., t_n) \Rightarrow \infty}$$

Fig. 7. Operational semantics of non-terminating expressions

on states which replaces the topmost mapping for r in S. A labeled expression is evaluated in an enriched state $S\!\uparrow$ where the current state is copied on top of the vector. The resulting state $S'\!\downarrow$ is obtained by deleting the second position of the vector what corresponds to "forget" the previously copied current state. The rule for function calls requires the pre-condition to be valid in the starting state and, if the function terminates normally, the validity of the post-condition in the returning state to be valid too.

Judgement $\Gamma, S, e \Rightarrow \infty$ expresses that the execution of expression e does not terminate in environment Γ and state S. Its definition is straightforward: the execution of an expression diverges if the execution of a sub-expression diverges. The interesting cases are for the execution of a loop: starting from a given state S, it diverges either if its body diverges or if its body terminates on some state S' and the whole loop diverges starting from this new state. Of course, non-termination may be caused by infinite recursion of functions, too.

The main feature to notice is that execution blocks whenever an invalid assertion is met: the rules for assertions, loops and function calls are applicable only if the respective annotations are valid. Conversely, as everything is well-typed by construction, the only reason why an expression wouldn't execute is that one of its annotations isn't respected.

Definition 2 (Safe execution). *An expression e* executes safely *in environment Γ and state S, denoted $\Gamma, S, e \overset{\text{safe}}{\Rightarrow}$, if either it diverges: $\Gamma, S, e \Rightarrow \infty$, or it terminates: $S', o,\ \Gamma, S, e \Rightarrow S', o$.*

A program respects its annotations *if for each function f and any Γ, S such that $[\![pre_f]\!]_{\Gamma,S}$ we have $\Gamma, S, body_f \overset{\text{safe}}{\Rightarrow}$ and if $\Gamma, S, body_f \Rightarrow S', o$ then o is a normal outcome v such that $[\![post_f]\!]_{v::\Gamma,S'}$.*

$$\begin{aligned}
\mathsf{WP}\ (\text{Eterm}\ t)\ Q\ R\ \Gamma\ S &= Q\ S\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Elet}\ e_1\ e_2)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e_1\ (\lambda S\ a,\ \mathsf{WP}\ e_2\ Q\ R\ (a :: \Gamma)\ S)\ R\ \Gamma\ S\\
\mathsf{WP}\ (\text{Eassign}\ r\ t)\ Q\ R\ \Gamma\ S &= Q\ (S[r/[\![t]\!]_{\Gamma,S}])\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Eassert}\ p\ e)\ Q\ R\ \Gamma\ S &= [\![p]\!]_{\Gamma,S} \wedge \mathsf{WP}\ e\ Q\ R\ \Gamma\ S\\
\mathsf{WP}\ (\text{Eraise}\ ex\ t)\ Q\ R\ \Gamma\ S &= R\ S\ ex\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Eif}\ p\ e_1\ e_2)\ Q\ R\ \Gamma\ S &= ([\![p]\!]_{\Gamma,S} \rightarrow \mathsf{WP}\ e_1\ Q\ R\ \Gamma\ S)\\
&\quad \wedge (\neg[\![p]\!]_{\Gamma,S} \rightarrow \mathsf{WP}\ e_2\ Q\ R\ \Gamma\ S)\\
\mathsf{WP}\ (\text{Eloop}\ p\ e)\ Q\ R\ \Gamma\ S &= [\![p]\!]_{\Gamma,S} \wedge \forall S', S \overset{\mathsf{writes}\ e}{\rightsquigarrow} S' \rightarrow [\![p]\!]_{\Gamma,S'} \rightarrow\\
&\qquad \mathsf{WP}\ e\ (\lambda S''\ v,\ [\![p]\!]_{\Gamma,S''})\ R\ \Gamma\ S'\\
\mathsf{WP}\ (\text{Etry}\ e_1\ ex\ e_2)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e_1\ Q\ (\lambda S'\ ex'\ a,\ \mathsf{if}\ ex = ex'\\
&\qquad \mathsf{then}\ \mathsf{WP}\ e_2\ Q\ R\ (a :: \Gamma)\ S'\ \mathsf{else}\ R\ ex'\ a)\ \Gamma\ S\\
\mathsf{WP}(\text{Elabel}\ e)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e\ Q\ R\ \Gamma\ S\!\uparrow\\
\mathsf{WP}\ (\text{Ecall}\ f\ (t_1, ..., t_n))\ Q\ R\ \Gamma\ S &= \ [\![pre_f]\!]_{\Gamma_{args},S} \wedge \forall S'\ a, S \overset{\mathsf{writes}\ f}{\rightsquigarrow} S' \rightarrow\\
&\qquad [\![post_f]\!]_{(a::\Gamma_{args}),(S',S)} \rightarrow Q\ S'\ a\\
&\quad \text{where } \Gamma_{args} := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}]
\end{aligned}$$

Fig. 8. Recursive definition of the WP-calculus

Our semantics is quite unusual, in particular it is not executable. Although, if annotations are removed then it becomes executable (indeed only if the propositional guards in if-then-else blocks are decidable) and coincides with a natural semantics. This approach makes obsolete a distinct set of rules for axiomatic semantics *à la* Hoare: the soundness of the verification condition generator will be stated using this definition of safe execution. Moreover this notion of safe execution is indeed stronger than the usual notion of partial correctness: a safe program that does not terminate will still satisfy its annotations forever.[1]

4 Weakest Precondition Calculus

We calculate the weakest pre-condition of an expression given a post-condition by structural recursion over expressions (Fig. 8). We admit several post-conditions, $Normal_{L,A} : state_L \rightarrow dentype_A \rightarrow \mathsf{Prop}$ for regular execution and $Exceptional_L : state_L \rightarrow \forall B, exn_B \rightarrow dentype_B \rightarrow \mathsf{Prop}$ for exceptional behavior. So our calculus has the type $\mathsf{WP} : e_{L,E,A} \rightarrow Normal_{L,A} \rightarrow Exceptional_L \rightarrow env_E \rightarrow state_L \rightarrow \mathsf{Prop}$. In the case of a loop, the pre-condition is calculated using the loop invariant and in the case of a function call we use the pre- and post-condition of that function. In both cases, as it is classical in WP calculi, we need to quantify over all states that may be reached by normal execution starting

[1] Total correctness is not considered in this paper; however it is clear that one could add annotations for termination checking: variants for loops and for recursive functions as in ACSL [4].

from the given state S: these are the states S' which differ from S only for the references that are modified in the loop or the function's body. This is denoted as $S \overset{s}{\leadsto} S' := \forall r : \mathit{ref}_A, r \notin s \rightarrow (\mathit{Here}\ S'\ r = \mathit{Here}\ S\ r \wedge \forall l, \mathit{At}\ l\ S'\ r = \mathit{At}\ l\ S\ r)$. The function *writes* computes the references modified by some expression, it is shown correct [16] in the sense that if $\Gamma, S, e \Rightarrow S', o$ then $S \overset{\mathit{writes}\ e}{\leadsto} S'$.

The verification conditions, respectively for one function and for a whole program, are

$$\begin{aligned} \mathsf{VC}(f) &:= [\![\mathit{pre}_f]\!]_{\Gamma,S} \rightarrow \mathsf{WP}\ \mathit{body}_f\ (\lambda S'\ v, [\![\mathit{post}_f]\!]_{v::\Gamma,S'})\ (\lambda S'\ \mathit{ex}\ v, \mathsf{False})\ \Gamma\ S \\ \mathsf{VCGEN} &:= \forall f : \mathit{idx}_{\langle A,P\rangle,F}\ \Gamma\ S,\ \mathsf{VC}(f) \end{aligned}$$

The False as exceptional post-conditions requires that no function body exits with an exception.

We now state that if the VCs hold for all functions then any expression having a valid WP executes safely. It is proved by co-induction, using the axiom of excluded middle to distinguish whether the execution of an expression does or does not terminate, following the guidelines of Leroy and Grall [20]. Notice that it is enough to prove the verification conditions for each function separately, even if functions can be mutually recursive, there is no circular reasoning.

Lemma 1 (safety of expressions). *If VCGEN holds then for any Γ, S, e, Q, R, if $(\mathsf{WP}\ e\ Q\ R\ \Gamma\ S)$ then $\Gamma, S, e \overset{\mathit{safe}}{\Rightarrow}$.*

The important corollary below states that if the VCs hold for all functions then any of their bodies execute safely. By definition of the semantics, this implies that all assertions, invariants and pre- and post-conditions in a given program are verified if the verification conditions are valid.

Theorem 1 (soundness of VCGEN). *If VCGEN holds then the program respects its annotations, as defined in Def. 2*

5 Extraction of a Certified Verification Tool

The obtained Coq function for generating verification conditions is not *extractable*: given a program pg we obtain a Coq term ($\mathsf{VCGEN}\ pg$) of Coq type Prop which must be proved valid to show the correctness of the program. The process thus remains based on Coq for making the proofs. In this section we show how to extract the calculus into a separate tool so that proofs can be performed with other provers, e.g. SMT solvers.

5.1 Concrete WP Computation

To achieve this we need the WP calculus to produce a formula in the abstract syntax of Fig. 2 instead of a Coq Prop. We define another function

$$\mathsf{wp} : e_{L,E,A} \rightarrow p_{L,A::E} \rightarrow (\mathit{exn}_B \rightarrow p_{L,B::E}) \rightarrow p_{L,E}$$

which, given an expression e, a normal post-condition Q and a family of exceptional post-conditions R, returns a weakest pre-condition. It is defined recursively on e similarly to WP in Fig. 8, but this time Q, R and the result are syntactic propositions which are concretely transformed (see [16]).

Lemma 2. *If* $[\![\mathsf{wp}\ e\ Q\ R]\!]_{\Gamma,S}$ *then*

$$\mathsf{WP}\ e\ \ (\lambda S\ v, [\![Q]\!]_{v::\Gamma,S})\ \ (\lambda S\ ex\ v, [\![R\ ex]\!]_{v::\Gamma,S})\ \ \Gamma\ S$$

From wp we now define a concrete verification-condition generator vcgen.

Definition 3. *The concrete VCs of a program pg is the list* $(\mathsf{vcgen}\ pg)$ *of concrete formulas* $(Abstr(\text{Pimply}\ pre_f\ \ (\mathsf{wp}\ body_f\ post_f\ \text{Pfalse})))$ *for each function f of pg. Abstr is a generalization function: it prefixes any formula $p_{L,E}$ by as many* Pforall *as elements of E to produce a $p_{L,[]}$ formula.*

Theorem 2. *If for all p in the list* $(\mathsf{vcgen}\ pg)$ *and for all state S,* $[\![p]\!]_{[],S}$ *then* $(\mathsf{VCGEN}\ pg)$*.*

That is, the hypothesis of Theorem 1 is valid if we prove the formulas generated by vcgen valid in any state.

5.2 Producing Concrete Syntax with Explicit Binders

Still, formulas of vcgen are represented by a de Bruijn-style abstract syntax. To print out such formulas we need to transform them into concrete syntax with identifiers for variables by generating new names for all the binders. This could be done on the fly in an unproven pretty-printer. Though, being a non trivial transformation it is better to do it in a certified way directly after the generation.

We therefore formalize a back-end syntax, along with its semantics for well-typed terms and propositions. It is similar to Fig. 2 where we replace Tconst, Tvar and Tderef by a new constructor Tvar with an identifier as argument, and Tlet and Pforall binders are also given an explicit identifier. We define a compilation from *de Bruijn*-style terms and propositions to the back-end syntax and prove preservation of semantics.

Finally, we define a *proof task* as a triple (d, h, g) where d is a finite map from identifiers to their type, h is a set of hypotheses and g is a list of goals. Such a task is said valid if the goals are logical consequences of the hypotheses, whatever the interpretation of symbols in d. The complete process of VC generation is to produce, from a logical context C and a program pg, the proof task $T(C, pg) = (d, h, g)$ where d are the declarations of C that appear in pg, h the compilation of axioms of C, and g is the compilation of $\mathsf{vcgen}(pg)$.

Theorem 3 (Main soundness theorem). *For all logical context C and program p, if the proof task $T(C, p)$ is valid then for any interpretation of the context in which the axioms are valid, p executes safely.*

Notice that this statement is independent of the underlying proof assistant Coq: the validity of logical formulas in the proof task can be established by any theorem prover. The only hypothesis is that the backend theorem prover in use must agree with our definition of the interpretation of logical contexts. But this is just the classical first-order logic with equality, with standard predefined theories like integer arithmetic. All the off-the-shelf theorem provers, e.g SMT solvers, agree on that.

5.3 Extraction and Experimentation

For experimentation purposes we also defined a compilation in the opposite direction, i.e. from programs in front-end syntax to the corresponding program in *de Bruijn* syntax, provided that the former is well typed. We then use the extraction mechanism of Coq to extract an Ocaml function that, given an AST of our front-end syntax representing a program, produces a list of ASTs representing the proof task. We finally combine this with the Why3 parser for input programs and a hand-written pretty-printer that produces Why3 syntax [6], allowing us to call automated provers on the proof task.

We made experiments to validate this process. On our selection sort example, the two VCs for functions `swap` and `selection_sort` are generated in a fraction of a second by the standalone VC generator. These are sent to the Why3 tool, and they are proved automatically, again in a fraction of a second, by a combination of SMT solvers (i.e. after splitting these formulas, which are conjunctions, into parts [6]). For details see the Coq development at the URL given in the introduction.

6 Conclusions, Related Works and Perspectives

We formalized a core language for deductive verification of imperative programs. Its operational semantics is defined co-inductively to support possibly non-terminating functions. The annotations are taken into account in the semantics so that validity of a program with respect to its annotations is by definition the progress of its execution. We used an original formalization of binders so that only well-typed programs can be considered, allowing us to simplify the rest of the formalization. Weakest precondition calculus is defined by structural recursion, even in presence of mutually recursive functions, assuming the given function contracts. Even if there is an apparent cyclic reasoning, this approach is shown sound by a co-inductive proof. By additionally formalizing an abstract syntax for terms and formulas, and relating their semantics with respect to the Coq propositions, we defined a concrete variant of the WP calculus which can be extracted to OCaml code, thus obtaining a trustable and executable VC generator close to Why or Boogie.

As explained in the introduction, two kinds of approaches for deductive verification exist depending on the use of a deep embedding of the programming language or not. The approaches without deep embedding typically allows the user to discharge proof obligations using automatic provers, but are not certified correct. Our work fills this gap. Among deep-embedding-based approaches, the SunRise system of Homeier et al. [17,18] is probably the first certified program verifier, and uses a deep embedding in the HOL proof environment. They formalize a core language and its operational semantics, and prove correct a set of Hoare-style deduction rules. Programs are thus specified using HOL formulas and proved within the HOL environment. Later Schirmer [26] formalized a core language in Isabelle/HOL, and Norrish formalized the C programming language [24], with similar approaches. More recently, similar deep-embedding-based approaches were proposed using Coq like in the Ynot system [23,9], which can deal with "pointer" programs via separation logic, and also supports higher-functions.

A major difference between the former approaches and ours is that we use a deep embedding not only for programs but also for propositions and thus for specifications. This allows us to extract a standalone executable, and consequently to discharge VCs using external provers like SMT solvers. Our approach is a basis to formalize specification languages like JML and ACSL defined on top of mainstream programming language, which allows a user to specify and prove Java or C programs without relying on the logic language of a specific proof assistant.

Another difference is that we do not consider any Hoare-style rules but formalize a Dijkstra-style VC generator instead. This way to proceed is motivated by the choice of defining the meaning of "a program satisfies its annotations" by safety of its execution.

There are also some technical novelties in our approach with respect to the systems mentioned above. Our core language supports exceptions, which is useful for handling constructs of front-ends like `break` and `continue`, or Java exceptions. Specifications can also use labels to refer to former states of executions, with constructs like `\old` and `\at` constructs of JML and ACSL. This provides a handy alternative to the so-called *auxiliary* or *ghost* variables used in deep-embedding-based systems above. Indeed in the context of VC generation instead of Hoare-style rules, the semantics of such variables is tricky, e.g. when calling a procedure, the ghost variables should be existentially quantified, which results in VCs difficult to solve by automated provers. We believe that the use of labels is thus better.

Our main future work is to certify the remaining part of a complete chain from ACSL-annotated C programs to proof obligations. A first step is the formalization of a front-end like Frama-C/Jessie which compiles annotated C to intermediate Why code. We plan to reuse the C semantics defined in CompCert [19] and incorporate ACSL annotations into it. The main issue in this compilation process is the representation of the C memory heap by Why global references using a memory heap modeling. In particular, first-order modeling of the heap, mainly designed to help automatic provers, raised consistency problems in the past [27]. In our approach where the axioms of the logical context are realized in Coq, the consistency is guaranteed. Finally, another part of the certification of the tool chain is the certification of back-end automatic provers, for which good progress was obtained recently, see e.g. [21].

Acknowledgments. We would like to thanks Jean-Christophe Filliâtre and Julien Signoles for their remarks on a preliminary version of this paper.

References

1. Abrial, J.-R.: The B-Book, assigning programs to meaning. Cambridge University Press (1996)
2. Barnett, M., DeLine, R., Jacobs, B., Chang, B.-Y.E., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009), `http://frama-c.cea.fr/acsl.html`
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
7. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)
8. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2011), `http://adam.chlipala.net/cpdt/`
9. Chlipala, A.J., Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: Hutton, G., Tolmach, A.P. (eds.) ICFP, pp. 79–90. ACM, Edinburgh (2009)
10. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting eSC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
11. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: 31st International Conference on Software Engineering Companion, ICSE 2009, Vancouver, Canada, May 16-24, pp. 429–430. IEEE Comp. Soc. Press (2009)
12. Filliâtre, J.-C.: Verification of non-functional programs using interpretations in type theory. Journal of Functional Programming 13(4), 709–745 (2003)
13. Filliâtre, J.-C., Marché, C.: Multi-Prover Verification of C Programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
14. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)

15. Herms, P.: Certification of a chain for deductive program verification. In: Bertot, Y. (ed.) 2nd Coq Workshop, Satellite of ITP 2010 (2010)
16. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. Research Report 7793, INRIA (2011), http://hal.inria.fr/hal-00639977/en/
17. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. The Computer Journal 38(2), 131–141 (1995)
18. Homeier, P.V., Martin, D.F.: Mechanical Verification of Mutually Recursive Procedures. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 201–215. Springer, Heidelberg (1996)
19. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
20. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. 207, 284–304 (2009)
21. Lescuyer, S.: Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq. Thèse de doctorat, Université Paris-Sud (2011)
22. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. Journal of Logic and Algebraic Programming 58(1-2), 89–106 (2004), http://krakatoa.lri.fr
23. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: Proceedings of ICFP 2008(2008)
24. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (November 1998)
25. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1798–1815. Springer, Heidelberg (1999)
26. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
27. Wagner, M., Bormer, T.: Testing a verifying compiler. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software, Papers Presented at the International Conference, Karlsruhe Reports in Informatics, Paris (2010), http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083

Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT[⋆]

Sabine Schmaltz and Andrey Shadrin

Saarland University, Germany
`{sabine,shavez}@wjpserver.cs.uni-saarland.de`

Abstract. Pervasive formal verification of operating systems and hypervisors is, due to their safety-critical aspects, a highly relevant area of research. Many implementations consist of both assembler and C functions. Formal verification of their correctness must consider the correct interaction of code written in these languages, which is, in practice, ensured by using matching application binary interfaces (ABIs). Also, these programs must be able to interact with hardware. We present an integrated operational small-step semantics model of intermediate-language C and *Macro-Assembler* code execution for pervasive operating systems and hypervisor verification. Our semantics is based on a compiler calling convention that defines callee- and caller-save registers. We sketch a theory connecting this semantic layer with an ISA-model executing the compiled code for use in a pervasive verification context. This forms a basis for soundness proofs of tools used in the VerisoftXT project and is a crucial step towards arguing formal correctness of execution of the verified code on a gate-level hardware model.

1 Introduction

For operating systems, correctness of implementation is highly desirable – in particular for safety-critical embedded systems such as cars or airplanes. Hypervisors are employed to partition system resources efficiently, providing strictly-separated execution contexts in which operating systems can again be run. To argue implementation correctness of a system consisting of both a hypervisor and operating systems, formal verification can provide solid evidence if done in a pervasive way.

The L4verified kernel [1] is an example of a recent operating system verification effort that has achieved impressive code verification results. A detailed memory model for low-level pointer programs in C was applied in combination with separation logic [2]. The assembler portions have not been verified in conjunction with the C code yet to our knowledge. Judging from their choice of C semantics, however, we are certain that all gaps present can be closed with minimal additional effort when the right models and theories are applied. In this paper, we present a description of such a theory for C and assembler code verification.

[⋆] Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

The FLINT group, on the other hand focuses on assembler code verification using their framework XCAP [3], which they successfully applied in [4] and [5]. So far, however, no integration of results into a semantics stack with high-level programming languages has been reported yet.

In a pervasive verification effort that aims at code verification above assembler-level, compiler correctness is crucial. During the Verisoft project, a compiler for the C-like language C0 was verified [6]. A mildly optimizing compiler that translates C-minor (a subset of C) to PowerPC assembly code has been verified by Xavier Leroy et al. [7,8] and used in a pervasive verification effort by Andrew Appel [9]. Both these efforts made use of interactive theorem provers.

In the scope of this paper we provide descriptions of the *Macro-Assembler* (short: *MASM*, section 2) and the C intermediate-language (*C-IL*, section 3) semantics we use to construct the integrated *C-IL+MASM*-semantics we propose in section 4. We combine a rather high-level assembler-semantics with a low-level C-intermediate-language semantics. This results in a model in which function calls between the two languages have the straightforward semantics we expect according to compiler calling conventions. We describe how we apply pervasive theory in section 5 to prove that our inter-language-call-semantics is sound with respect to the underlying machine-code execution model. Note that all of this can still be considered work-in-progress since none of the proofs have been checked in a theorem prover.

As main contributions of this paper, we consider, first, our unconventional choice to design both *C*- and assembler-semantics in such a way that they can interoperate easily – resulting in a model that accurately captures the compiler calling convention –, and, second, our demonstration that such an integrated model can be easily justified using pervasive compiler correctness theory.

2 Macro-Assembler Semantics

One might wonder why, in a pervasive verification effort, there is any need for a high-level assembler semantics. Operating systems and hypervisors implemented in C and assembler are generally compiled to machine code – a machine-code execution model technically is fully sufficient to argue about such systems. However, doing this, we would discard all the comfort and gain of speed that appropriate abstraction can provide. We consider a machine-code execution model that we refer to using the name *ISA-Assembler*. It is characterized by the following: Instructions are executed as atomic transitions – an instruction pointer register points to the next instruction in memory.

Concerning code verification, the *ISA-Assembler*-model has one particular drawback: It provides no useful abstraction in terms of control flow. While this is true for the language of machine code, the assembler code used in operating systems has structure we can exploit during formal verification: Our assembler code is called from or calls functions of a stack-based programming language. Instead of executing machine code instructions from the configuration's memory at the instruction pointer, we apply abstraction techniques normally used in high-level programming language semantics definitions. We gain an easy-to-understand model of high-level assembler code execution that can be integrated with a C model in a straightforward way. This comes at a cost: In order to obtain a more simple model of assembler code execution, we enforce a certain structure of code which may exclude well-behaving assembler programs. However, all code we want to verify has this structure.

We introduce *Macro-Assembler* (*MASM*) as a restricted assembler language: All targets of branch or jump instructions are either local labels or names of functions – we model the control flow of *Macro-Assembler* as a labeled transition system. In order to make integration of *MASM* with a stack-based programming language very simple, we abstract from the concrete stack layout in memory by introducing a stack component in form of a list of abstract stack frames to the configuration.

In [10], a stack-based typed low assembly language is proposed as a target language for code verification. The authors can encode any compiler calling conventions in their type system since everything about the stack including the stack frame header layout is exposed. In our work we abstract the stack away hiding all details that are compiler relevant. Additionally, we provide a feature found in some assembler languages: uses lists that specify the registers used by an assembler function. The *MASM*-compiler inserts instructions that save/restore these registers in the prologue/epilogue of the compiled assembler function. In the following, we present formal definitions to elaborate on the structure of *MASM*.

Configuration. A *Macro-Assembler* configuration

$$c = (c.\mathcal{M}, c.\mathit{regs}, c.s) \in \mathit{conf}_{\mathit{MASM}}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}^{8}$ (where k is the number of bytes in a machine word and $\mathbb{B} \equiv \{0, 1\}$), a component $\mathit{regs} : \mathcal{R} \to \mathbb{B}^{8k}$ that maps register names to their values, and an abstract stack $s : \mathit{frame}^{*}_{\mathit{MASM}}$. Each frame

$$s[i] = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})$$

contains the name p of the assembler function we are executing in, the location *loc* of the next instruction to be executed in p's body, a component *saved* that is used to store values of callee-save registers used by the function, a component *pars* that represents the parameter region of the stack frame, and a component *lifo* that represents the part of the stack where data can be $push$ed and popped to/from.

Program. A *Macro-Assembler* program π is a procedure table that maps function names p to procedure table entries:

$$\pi(p) = (\mathit{npar}, \mathcal{P}, \mathit{uses})$$

Here, *npar* describes the total number of machine word parameters of the function, $\mathcal{P} : \mathit{instr}^{*}_{\mathit{MASM}}$ is a list of *Macro-Assembler* instructions representing the procedure body, and *uses*: $\mathcal{R}^{*}$ is a list of register names to be saved and restored.

In case the following instructions are not provided by the underlying hardware, we implement them as assembler macros: *call*, *ret*, *push*, *pop*. An assembler macro is simply a shorthand for a sequence of assembler instructions. *MASM* can easily be extended by the notion of user-defined macros, however, we have not done so yet.

Table 1. Operational semantics of *MASM*

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{instr}(i) \qquad \mathit{MASM\text{-}to\text{-}ISA}(c) \underset{\mathit{ISA}}{\rightarrow} d'}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{ISA\text{-}to\text{-}MASM}(d')} \; \text{(INSTR)} \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{goto}\ l}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \; \text{(GOTO)}$$

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{ifnez}\ r\ \mathbf{goto}\ l \qquad c.\mathit{regs}(r) = 0^{8k}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{inc}_{\mathit{loc}}(c)} \; \text{(GOTO-FAIL)} \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{ifnez}\ r\ \mathbf{goto}\ l \qquad c.\mathit{regs}(r) \neq 0^{8k}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \; \text{(GOTO-SUCC)}$$

$$\frac{\begin{array}{c}\mathit{instr}_{\mathit{next}}(c) = \mathbf{push}\ r \\ \mathbf{hd}(c.s) = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})\end{array}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{lifo}}(\mathit{inc}_{\mathit{loc}}(c), c.\mathit{regs}(r) \circ \mathit{lifo})} \; \text{(PUSH)} \qquad \frac{\begin{array}{cc}\mathit{instr}_{\mathit{next}}(c) = \mathbf{call}\ p & \pi(p).\mathit{npar} - 4 \leq \mathbf{hd}(c.s).\mathit{lifo} \\ \mathit{call}_{\mathit{frame}}(c, p, \mathit{frame}_{\mathit{new}}) & c' = \mathit{drop}_{\mathit{lifo}}(c, \pi(p).\mathit{npar} - 4)\end{array}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} c'[s := \mathit{frame}_{\mathit{new}} \circ \mathit{inc}_{\mathit{loc}}(c').s]} \; \text{(CALL)}$$

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{pop}\ r \qquad \mathit{lifo} \neq [] \qquad \mathbf{hd}(c.s) = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{lifo}}(\mathit{set}_{\mathit{reg}}(c, r, \mathbf{hd}(\mathit{lifo})), \mathbf{tl}(\mathit{lifo}))} \; \text{(POP)} \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \mathbf{ret}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{drop}_{\mathit{frame}}(\mathit{restore}_{\mathit{saved}}(c))} \; \text{(RET)}$$

Semantics. Since *Macro-Assembler* is compiled to machine code, *Macro-Assembler* semantics for basic instructions can be inferred from the semantics of *ISA-Assembler*. The main differences stem from the distinct modeling of control-flow: In *ISA-Assembler*, we have a global instruction pointer whereas in *Macro-Assembler* we have local program location and stack-abstraction. Whenever a *Macro-Assembler* instruction accesses the stack-region, instead of updating/reading the memory, we update/read the corresponding component of the abstract stack. In case it is not trivially possible to find an equivalent update on the abstract stack, we consider the program in question illegal – or unsuitable for use with our semantics. This concerns, in particular, all instructions that explicitly update the stack pointer registers (this would break our stack abstraction) or write to addresses of the physical stack that describe return addresses or previous frame base addresses.

In the *call*-rule presented in table 1, $\mathit{call}_{\mathit{frame}}(c, p, \mathit{frame}_{\mathit{new}})$ is a predicate over a configuration c, a function p, and a frame f that enforces following conditions:

$$\mathit{frame}_{\mathit{new}}.p = p, \quad \mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \quad \mathit{frame}_{\mathit{new}}.\mathit{saved} = c.\mathit{regs}|_{\pi(p).\mathit{uses}}$$

$$\mathit{frame}_{\mathit{new}}.\mathit{pars}[\pi(p).\mathit{npar} - 1 : 4] = \mathit{read}_{\mathit{lifo}}(c, \pi(p).\mathit{npar} - 4), \quad \mathit{frame}_{\mathit{new}}.\mathit{lifo} = []$$

All registers in the uses list of the function are saved in the new frame, and parameters that are passed on the stack are moved from the *lifo*-component of the top-most frame to the *pars*-component of the new frame. The calling convention we assume states that the first four parameters are passed in registers (space on the stack is reserved nonetheless), while the remaining parameters are passed on the stack (in right-to-left order).

3 C Semantics

As countless others have noted before [11,12], there is not "the" C semantics: the term "C" describes an equivalence class of semantics that fall under the scope of what is

Table 2. The set $\mathbb{T}$ of types of *C-IL*

$t \in \mathbb{T}$	$t \in \mathbb{T}_P$
struct $t_C \in \mathbb{T}$	$t_C \in \mathbb{T}_C$
array$(t, n) \in \mathbb{T}_{\textbf{ptr}}$	$t \in \mathbb{T}, n \in \mathbb{N}$
ptr$(t) \in \mathbb{T}_{\textbf{ptr}}$	$t \in \mathbb{T}$
fptr$(t, T) \in \mathbb{T}_{\textbf{ptr}}$	$t \in \mathbb{T}, T \in \mathbb{T}^*$

Table 3. The set val of values of *C-IL*

val$(b, \mathbf{i}i) \in val$	$b \in \mathbb{B}^i$
val$(b, \mathbf{u}i) \in val$	$b \in \mathbb{B}^i$
val$(B, \textbf{struct } t_C) \in val_{\textbf{struct}}$	$B \in (\mathbb{B}^8)^*$
val$(b, t) \in val_{\textbf{ptr}}$	$b \in \mathbb{B}^{size_{ptr}}, t \in \mathbb{T}_{\textbf{ptr}}$
lref$((v, o), i, t) \in val_{\textbf{lref}}$	$v \in \mathbb{V}, o, i \in \mathbb{N}, t \in \mathbb{T}_{\textbf{ptr}}$
fun$(f) \in val_{\textbf{fun}}$	$f \in \mathbb{F}_{name}$

commonly called the programming language C and its standard library. Depending on architecture and compiler, semantics may differ for the underspecified areas.

Since C is a programming language with an overwhelming complexity – much of which is redundant –, we consider an intermediate language for C that we call *C-IL*. Note that, instead of defining Pascal with C syntax (as has been done in Verisoft), we now consider a semantics that really captures the low-level features of C. We do not consider side-effects in expressions, and neither do we put much effort on modeling C syntax. These, we leave to the layer above, where we can define C based on *C-IL*.

The intermediate language we consider has been designed with some very specific features – optimized for integration with *Macro-Assembler*. Like *MASM*, *C-IL* is a goto-language defined in the form of a labeled transition system.

Since we want to do lowest-level operating systems verification, we only consider a global byte-addressable memory and an abstract stack. We do not consider the heap as a separate memory since the notion of a heap only exists when there is some form of memory allocation system available (e.g. the one provided by the standard library or the operating system). Pointer arithmetics is allowed on pointers to the global memory to the full extent possible. For local variables we restrict pointer arithmetics to calculating offsets inside local variable memories. In the semantics we propose, every memory access corresponds to dereferencing a left value – a left value is either a pointer to the global memory or a reference to some offset in a local variable.

In interrupt descriptor tables, we need to store function pointer values, thus, we explicitly model the addresses of functions. Obviously, these cannot be derived from C semantics since they depend only on where in memory the program resides, thus we give them as parameter to the semantics.

One main issue of C is its dependency on the underlying architecture and compiler. We suggest that semantics for C should be parameterizable to make it applicable to at least the most common cases.

In the following, $\mathbb{V}$ is a set of variable names, $\mathbb{F}$ is a set of field names, $\mathbb{F}_{name}$ is a set of function names. We use the notation $\mathcal{X}^* \equiv \bigcup_{n=1}^{\infty} \mathcal{X}^n \cup \{[]\}$ to describe the set of lists/strings with elements from the set $\mathcal{X}$. A list of length n with elements from $\mathcal{X}$ is given by $x = (x_{n-1}, \ldots, x_0) = x[n-1:0] \in \mathcal{X}^n$ and we define the shorthand $x[i] \overset{def}{=} x_i$.

Types. For every instance of *C-IL*, we assume a set of primitive types $\mathbb{T}_P$ to be given such that $\mathbb{T}_P \subset \{\textbf{void}\} \cup \bigcup_{i=0}^{\infty} \{\mathbf{i}i, \mathbf{u}i\}$ describes a set of basic signed ($\mathbf{i}$) and unsigned (**u**) integer types of size i (given in bits). Usually, we consider sizes which are multiples of 8. Further, we assume a set $\mathbb{T}_C$ of struct type names to be given.

We define the set of types $\mathbb{T}$ inductively in table 2. Array types are given by their element type and length. Function pointer types are identified by their return value type and a list of their parameters' types. Note that struct types are always identified by the corresponding composite type name t_C. The actual type definition of a struct type can be found by looking it up in the program's struct type declaration (defined later).

Values. We represent most values using bit-strings or byte-strings. This is owed to the fact that *C-IL* is designed for use in conjunction with hardware models. For struct types, we consider byte-strings as their value (see table 3). We only need them in order to model struct assignment since access to a field of a struct is performed by calculating the corresponding left value followed by a precise memory access. A pointer to the global memory is a value $\textbf{val}(b, t)$ consisting of an address and a pointer type.

Due to the stack abstraction used, we have to treat pointers to local variables differently: We represent these *local references* $\textbf{lref}((v, o), i, t)$ by variable name v and offset o inside that variable. Additionally, we have the number i of the stack frame the local reference refers to and a pointer type t.

For functions f we do not need the exact address of (since we do not need to store their function pointers in memory), we introduce a symbolic function value $\textbf{fun}(f)$.

Expressions. We define the set of expressions $\mathbb{E}$ in table 4. $\mathbb{O}_1$ and $\mathbb{O}_2$ are sets of operators (table 5) defined for the compiler in question. A unary operator is a partial function $\oplus : val \rightharpoonup val$, whereas a binary operator is a function $\oplus : val \times val \rightharpoonup val$. Operators are provided for each type they are meaningful for. All expressions are strictly typed in *C-IL* – when translating from *C* to *C-IL*, type casts need to be inserted explicitly.

Statements. *C-IL* uses a reduced set of statements (see table 6) consisting of assignment, goto, if-not-goto, function call, procedure call, and corresponding return statements. Goto statements specify the target destination in form of a label (the index of the target statement in the program).

Configuration. A *C-IL* configuration

$$c = (\mathcal{M}, s) \in \mathit{conf}_{\text{C-IL}}$$

consists of a global, byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}^8$ and a stack $s \in \mathit{frame}^*_{\text{C-IL}}$ which is a list of *C-IL*-frames. A *C-IL*-frame

$$s[i] = (\mathcal{M}_{\mathcal{E}}, \mathit{rds}, f, \mathit{loc}) \in \mathit{frame}_{\text{C-IL}}$$

consists of a local memory component $\mathcal{M}_{\mathcal{E}} : \mathbb{V} \to (\mathbb{B}^8)^*$ which maps variable names to local byte-offset-addressable memories (represented as lists of bytes), a return destination component $\mathit{rds} : \mathit{val}_{\textbf{ptr}} \cup \mathit{val}_{\textbf{lref}} \cup \{\bot\}$ which is either a pointer to where the return value of the function is going to be stored when it returns or the value $\bot$ denoting the absence of a return destination, a function name f which describes the function we are executing and a location $\mathit{loc} \in \mathbb{N}$ which describes where in f's body execution should continue.

Table 4. The set $\mathbb{E}$ of *C-IL*-expressions

constants	c	$c \in val$
variable names	v	$v \in \mathbb{V}$
function names	f	$f \in \mathbb{F}_{name}$
unary operation	$\oplus e$	$e \in \mathbb{E}$ and $\oplus \in \mathbb{O}_1$
binary operation	$(e_1 \oplus e_2)$	$e_1, e_2 \in \mathbb{E}$ and $\oplus \in \mathbb{O}_2$
ternary operation	$(e \; ? \; e_1 : e_2)$	$e, e_1, e_2 \in \mathbb{E}$
type cast	$(t)e$	$t \in \mathbb{T}$ and $e \in \mathbb{E}$
dereferencing	$*e$	$e \in \mathbb{E}$
address-of	$\&e$	$e \in \mathbb{E}$
field access	$(e).f$	$e \in \mathbb{E}$ and $f \in \mathbb{F}$
size of type	**sizeof**(t)	$t \in \mathbb{T}$
size of expression	**sizeof**(e)	$e \in \mathbb{E}$

Table 5. Operators of *C-IL*

unary operators	$\mathbb{O}_1 = \{$`-`, `~`, `!`$\}$
binary operators	
$\mathbb{O}_2 = \{$`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `|`, `^`, `&&`, `||`$\}$	

Table 6. The set $\mathbb{S}$ of *C-IL*-statements

$e_0 = e_1$	$e_0, e_1 \in \mathbb{E}$
goto l	$l \in \mathbb{N}$
ifnot e **goto** l	$e \in \mathbb{E}, l \in \mathbb{N}$
$e_0 =$ **call** $e(E)$	$e_0, e \in \mathbb{E}, E \in \mathbb{E}^*$
call $e(E)$	$e \in \mathbb{E}, E \in \mathbb{E}^*$
return, **return** e	$e \in \mathbb{E}$

Program. A *C-IL* program

$$\pi = (\mathcal{F}, \mathcal{V}_G, T_F)$$

consists of a function table $\mathcal{F}$, a declaration of global variables $\mathcal{V}_G : (\mathbb{V} \times \mathbb{T})^*$ consisting of pairs of variable names and types, and a struct type declaration $T_F : \mathbb{T}_C \to (\mathbb{F} \times \mathbb{T})^*$ which returns for every composite type name a declaration of its fields.

A function table entry

$$\pi.\mathcal{F}(f) = (npar, \mathcal{V}, \mathcal{P})$$

contains the number of parameters $npar$ of the function f, a local variable and parameter declaration $\mathcal{V} : (\mathbb{V} \times \mathbb{T})^*$ and a function body $\mathcal{P} : \mathbb{S}^*$.

Context. Configuration and program are not enough: we need additional information in order to execute a *C-IL* program. For this, we introduce a context θ which provides all missing information. It contains information on the endianness of the underlying architecture (i.e. byte-order used), the addresses of global variables in memory, function pointer addresses (given by a partial, injective function $\theta.\mathcal{F}_{adr}$), offsets of fields in struct types, sizes of struct types, a type-casting function that matches the behavior of the compiler, and the type used by the compiler for results of the `sizeof`-operator.

Expression Evaluation. Expressions are evaluated by a function that returns either a *C-IL*-value or the special value $\perp$ that denotes that the expression cannot be evaluated:

$$[e]_c^{\pi,\theta} \in val \cup \{\perp\}$$

Depending on the expression, we may need the complete state, i.e. configuration, program and context, to evaluate it. Since expression evaluation is defined in the obvious way, given the choices we made, we omit its definition to save space.

Memory Semantics. On the one hand we have byte-addressable memories, on the other we have typed values. We provide functions $read^\theta : conf_{C\text{-}IL} \times (val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}) \to val$ and $write^\theta : conf_{C\text{-}IL} \times val \times (val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}) \to conf_{C\text{-}IL}$ which, respectively,

Table 7. Operational semantics of *C-IL*

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = e_0 = e_1}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{inc}_{\mathit{loc}}(\mathit{write}^{\theta}(c, [\&e_0]_c^{\theta,\pi}, [e_1]_c^{\theta,\pi}))}\ (\textsc{Assign}) \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{goto}\ l}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)}\ (\textsc{Goto})$$

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \qquad \textbf{zero}([e]_c^{\theta,\pi})}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)}\ (\textsc{IfNotGoto-Succ}) \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \qquad \neg\textbf{zero}([e]_c^{\theta,\pi})}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{inc}_{\mathit{loc}}(c)}\ (\textsc{IfNotGoto-Fail})$$

$$\frac{\begin{array}{c}\mathit{stmt}_{\mathit{next}}(c) = \textbf{call}\ e(E) \vee \mathit{stmt}_{\mathit{next}}(c) = e_0 = \textbf{call}\ e(E) \\ \mathit{is\text{-}function}([e]_c^{\theta,\pi}, f) \qquad \mathit{call}_{\mathit{frame}}(c, f, E, \mathit{frame}_{\mathit{new}})\end{array}}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} c[s := \mathit{frame}_{\mathit{new}} \circ \mathit{inc}_{\mathit{loc}}(c).s]}\ (\textsc{Call}) \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return}}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{drop}_{\mathit{frame}}(c)}\ (\textsc{Return})$$

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return}\ e \qquad c.\mathit{rds}_{\mathit{top}} \neq \bot}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{write}^{\theta}(\mathit{drop}_{\mathit{frame}}(c), c.\mathit{rds}_{\mathit{top}}, [e]_c^{\theta,\pi})}\ (\textsc{ReturnVal1}) \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return}\ e \qquad c.\mathit{rds}_{\mathit{top}} = \bot}{\pi, \theta \vdash c \underset{C\text{-}IL}{\rightarrow} \mathit{drop}_{\mathit{frame}}(c)}\ (\textsc{ReturnVal2})$$

dereference a pointer value in a given configuration (read from memory) or write a given value to memory, resulting in a new configuration. To specify their effect, similar functions ($\mathit{read}^{\theta}_{\mathcal{E}}$, $\mathit{write}^{\theta}_{\mathcal{E}}$) are provided to read and write a local variable/parameter (identified by variable name) from a stack frame.

Note that, since we do not model addresses of local variables explicitly (this would either expose stack layout or require a more sophisticated memory), our semantics carries the limitation that pointers to local variables cannot be stored in memory.

Operational Semantics. In table 7, we give operational semantics of *C-IL*. **zero** is a predicate that is true when the given *C-IL*-value is a representation of zero. The next statement to be executed in the given configuration is computed by $\mathit{stmt}_{\mathit{next}}$ from program/location of the top-most stack frame. With $\mathit{inc}_{\mathit{loc}}$ and $\mathit{drop}_{\mathit{frame}}$ we produce configurations in which, respectively, the location counter of the top-most frame is incremented or the top-most frame is simply removed from the stack.

In the *call*-rule, the new stack frame $\mathit{frame}_{\mathit{new}}$ is chosen nondeterministically according to the following constraints (represented by the predicate $\mathit{call}_{\mathit{frame}}(c, f, E, \mathit{frame}_{\mathit{new}})$):

$$\forall 0 \leq i < \mathit{npar} : \quad \mathit{read}^{\theta}_{\mathcal{E}}(\mathit{frame}_{\mathit{new}}, v_i, 0, t_i) = [E[i]]_c^{\theta,\pi}$$

$$\forall \mathit{npar} \leq i < \mathbf{len}(\mathcal{V}) : \quad \mathbf{len}(\mathit{frame}_{\mathit{new}}.\mathcal{M}_{\mathcal{E}}(v_i)) = \mathit{size}(t_i)$$

$$\mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \quad \mathit{frame}_{\mathit{new}}.f = f, \quad \mathit{frame}_{\mathit{new}}.\mathit{rds} = \begin{cases} [\&e_0]_c^{\theta,\pi} & \text{for function call} \\ \bot & \text{for procedure call} \end{cases}$$

Here, $(v_i, t_i) = \mathcal{V}[i]$ is the i-th declaration in function f's parameters and local variable declaration $\mathcal{V} = \pi.\mathcal{F}(f).\mathcal{V}$, and $\mathit{npar} = \pi.\mathcal{F}(f).\mathit{npar}$ is the number of parameters of the function f. Note that we only place a strict constraint on the parameter values: initial content of local variables is chosen nondeterministically with appropriate size for the declared type.

4 Integrated *C-IL+MASM*-Semantics

We extend both *C-IL* and *MASM* in such a way that we can do the final step of integrating them into *C-IL+MASM*. To achieve this, we define a compiler calling convention and apply it to interface the two languages. The goal of this integration is to 'slice' the model stack horizontally, providing a self-contained model to argue about a system layer that involves both *C-IL* and *MASM* code execution.

In the first Verisoft project, whenever assembler code is encountered, the compiler simulation relation is applied to reach an equivalent *ISA-Assembler*-configuration from which to execute the assembler code. The proposed integrated semantics simply provides another layer of abstraction on top of such a model. The assembler verification approach that was used in the VerisoftXT project is based on translating assembler code to C so that it can be verified using a C verification tool [13]. We can benefit from the abstraction we introduce here in a soundness proof for the assembler verification approach: Instead of proving a simulation between *ISA-Assembler* and *C-IL* (which would require a substantial amount of software conditions), we can prove a simpler simulation between *MASM* and *C-IL*. In turn, we have to prove correct compilation for *MASM*.

There is a restriction on the interaction between *C-IL* and *MASM*: currently, we only allow primitive values to be passed between *C-IL*- and *MASM*-functions.

4.1 Calling Convention

The compiler calling convention describes the interface between the caller and the callee. In our experiments, we consider a compiler calling convention given by the following rules:

1. The first four parameters are passed through Registers $R_{p_1}, R_{p_2}, R_{p_3}, R_{p_4}$.
2. The remaining parameters (if existent) are passed on the stack in right-to-left order. There is space reserved on the stack for parameters passed in registers.
3. The return value is passed through register R_{rv}.
4. All callee-save registers (given by $\mathcal{R}_{\textbf{callee}} \subset \mathcal{R}$) must be restored before return.
5. The callee is responsible for cleaning up the stack.
6. $R_{p_1}, R_{p_2}, R_{p_3}, R_{p_4}, R_{rv} \notin \mathcal{R}_{\textbf{callee}}$.

4.2 Semantics

In order to obtain an integrated model of *C-IL* and *MASM*, there are two things left to do: define how we model the state of the combined semantics and define transitions.

Probably the most basic way to define a configuration of *C-IL+MASM* is to consider a list of alternating *C-IL*- and *MASM*-configurations that represents the call stack between the two languages. The top-most configuration we consider *active* while we consider the rest of them *inactive*. One observation that can be made is that in both semantics we use the same byte-addressable memory, which can be shared.

In order to eliminate redundancy, we introduce the notion of *execution context* for *C-IL* and *MASM*. An *execution context* is a configuration of the corresponding language where the memory component $\mathcal{M}$ is removed:

$$s \in \mathit{context}_{\mathit{C\text{-}IL}} \equiv \mathit{frame}^*_{\mathit{C\text{-}IL}}, \qquad (\mathit{regs}, s) \in \mathit{context}_{\mathit{MASM}} \equiv (\mathcal{R} \to \mathbb{B}^{8k}) \times \mathit{frame}^*_{\mathit{MASM}}$$

Another observation we make is that in order to integrate the two semantics, we need to add information to inactive *C-IL*-execution-contexts: It would be very nice to know where the *C-IL*-execution context will store the return value that is passed in R_{rv} when it becomes active again. Another notion we want to capture in the semantics is that the *C-IL*-compiler may rely on the callee-save convention being respected by the programmer: When the callee-save registers have modified values, there is no guarantee whatsoever that execution of the *C-IL*-code will continue as expected. We define the inactive *C-IL*-execution-context

$$(\mathit{rds}, \mathit{regs}_{\mathbf{callee}}, s) \in \mathit{context}^{\mathit{inactive}}_{\mathit{C\text{-}IL}}$$

which consists of a return destination pointer $\mathit{rds} \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$, a function $\mathit{regs}_{\mathbf{callee}} : \mathcal{R}_{\mathbf{callee}} \rightharpoonup \mathbb{B}^{8k}$ which describes the content of callee-save registers expected when control is returned to the execution context, and a *C-IL*-stack $s \in \mathit{frame}^*_{\mathit{C\text{-}IL}}$.

The last observation is that it is not meaningful to store register values in the inactive *MASM*-execution-context, with one exception: we can keep the values of callee-save registers, since, assuming the *C-IL*-compiler respects the calling conventions, they will be restored when control is returned to the context. We define the inactive *MASM*-execution-context

$$(\mathit{regs}_{\mathbf{callee}}, s) \in \mathit{context}^{\mathit{inactive}}_{\mathit{MASM}}$$

to consist of a callee-save register file $\mathit{regs}_{\mathbf{callee}} : \mathcal{R}_{\mathbf{callee}} \to \mathbb{B}^{8k}$ that holds the values of callee-save registers belonging to the execution context, and a *MASM*-stack $s \in \mathit{frame}^*_{\mathit{MASM}}$.

Configuration. A *C-IL+MASM*-configuration

$$c = (\mathcal{M}, ac, sc) \in \mathit{conf}_{\mathit{C\text{-}IL+MASM}}$$

consists of the same byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}$ we have seen before, the active execution context $ac : \mathit{context}_{\mathit{C\text{-}IL}} \cup \mathit{context}_{\mathit{MASM}}$, and a list of inactive execution contexts $sc \in (\mathit{context}^{\mathit{inactive}}_{\mathit{C\text{-}IL}} \cup \mathit{context}^{\mathit{inactive}}_{\mathit{MASM}})^*$.

Program and Context. A *C-IL+MASM*-program $\pi = (\pi_{\mathit{C\text{-}IL}}, \pi_{\mathit{MASM}})$ is simply a pair of a *C-IL*-program and a *MASM*-program. Since we need context information about the compiler to execute *C-IL*, we keep the context θ from *C-IL*.

Transitions. Essentially, we have three types of steps: We perform a pure *C-IL*- or *MASM*-step, an external function from the other language is called, or we return to the other language. Considering a given configuration, it is easy to decide which of these has to happen next: Calling a function which is not declared in the current language's program must be an external call. Executing a return-statement or -instruction when the stack of the active context is empty should return to the newest context from the list of inactive contexts. Everything else is a pure step. Let $\mathit{ext}(c)$ denote a predicate that

Table 8. Representative choice of transitions from the semantics of *C-IL+MASM*

$$\frac{c.ac = s \qquad \neg ext(c) \qquad \pi.\pi_{\mathit{C\text{-}IL}}, \theta \vdash (c.\mathcal{M}, s) \rightarrow_{\mathit{C\text{-}IL}} c'_{\mathit{C\text{-}IL}}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, \mathcal{M} := c'_{\mathit{C\text{-}IL}}.\mathcal{M},\ ac := c'_{\mathit{C\text{-}IL}}.s \,\right]} \quad (\textsc{Pure-C-IL})$$

$$\frac{c.ac = (\mathit{regs}, s) \qquad \neg ext(c) \qquad \pi.\pi_{\mathit{MASM}} \vdash (c.\mathcal{M}, \mathit{regs}, s) \rightarrow_{\mathit{MASM}} c'_{\mathit{MASM}}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, \mathcal{M} := c'_{\mathit{MASM}}.\mathcal{M},\ ac := (c'_{\mathit{MASM}}.\mathit{regs}, c'_{\mathit{MASM}}.s) \,\right]} \quad (\textsc{Pure-MASM})$$

$$\frac{\begin{array}{c} c.ac = s \qquad ext(c) \qquad \mathit{stmt}_{\mathit{next}}(s, \pi.\pi_{\mathit{C\text{-}IL}}) = e_0 = \textbf{call}\ e(E) \\ \mathit{is\text{-}function}([e]_c^{\theta,\pi}, f) \qquad \mathit{CIL2MASM}_{\mathit{ctxt}}(c, f, E, \mathit{regs}_{\textbf{callee}}, \mathit{context}_{\mathit{new}}) \end{array}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, ac := \mathit{context}_{\mathit{new}},\ sc := ([\![\&e_0]\!]_c^{\theta,\pi}, \mathit{regs}_{\textbf{callee}}, \mathit{inc}_{\mathit{loc}}(s)) \circ c.sc \,\right]} \quad (\textsc{C-IL-to-MASM})$$

$$\frac{\begin{array}{c} c.ac = (\mathit{regs}, s) \qquad ext(c) \qquad \mathit{instr}_{\mathit{next}}(s, \pi.\pi_{\mathit{MASM}}) = \textbf{call}\ p \\ \pi.\pi_{\mathit{C\text{-}IL}}(p).\mathit{npar} - 4 \leq \textbf{hd}(s).\mathit{lifo} \qquad s' = \mathit{drop}_{\mathit{lifo}}(s, \pi(p).\mathit{npar} - 4) \qquad \mathit{MASM2CIL}_{\mathit{ctxt}}(c, p, \textbf{hd}(s).\mathit{lifo}, \mathit{context}_{\mathit{new}}) \end{array}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, ac := \mathit{context}_{\mathit{new}},\ sc := (\mathit{regs}|_{\mathcal{R}_{\textbf{callee}}}, \mathit{inc}_{\mathit{loc}}(s')) \circ c.sc \,\right]}$$
(MASM-TO-C-IL)

$$\frac{\begin{array}{c} c.ac = s \qquad ext(c) \qquad \mathit{stmt}_{\mathit{next}}(s, \pi.\pi_{\mathit{C\text{-}IL}}) = \textbf{return}\ e \\ \textbf{hd}(c.sc) = (\mathit{regs}_{\textbf{callee}}, s') \qquad \mathit{regs}'|_{\mathcal{R}_{\textbf{callee}}} = \mathit{regs}_{\textbf{callee}} \qquad \mathit{regs}'(R_{rv}) = \mathit{val2bytes}([\![e]\!]_c^{\pi,\theta}) \end{array}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, ac := (\mathit{regs}', s'),\ \ sc := \textbf{tl}(c.sc) \,\right]}$$
(RETURN-C-IL-TO-MASM)

$$\frac{\begin{array}{c} c.ac = (\mathit{regs}, s) \qquad ext(c) \qquad \mathit{instr}_{\mathit{next}}(s, \pi.\pi_{\mathit{MASM}}) = \textbf{ret} \\ \textbf{hd}(c.sc) = (\mathit{rds}, \mathit{regs}_{\textbf{callee}}, s') \qquad \mathit{regs}|_{\mathit{dom}(\mathit{regs}_{\textbf{callee}})} = \mathit{regs}_{\textbf{callee}} \qquad c' = \mathit{write}^{\theta}_{\mathit{raw}}((c.\mathcal{M}, s'), \mathit{rds}, \mathit{regs}(R_{rv})) \end{array}}{\pi, \theta \vdash c \rightarrow_{\mathit{C\text{-}IL+MASM}} c\left[\, \mathcal{M} := c'.\mathcal{M},\ ac := c'.s,\ \ sc := \textbf{tl}(c.sc) \,\right]}$$
(RETURN-MASM-TO-C-IL)

checks in the described way whether the next step is an external step. Table 8 shows inference rules that describe *C-IL+MASM*'s transitions.

Pure Steps. For $\neg ext(x)$, we have a pure step: We simply perform a step of the top-most execution context according to the semantics of the corresponding language.

Call from C-IL to MASM. For an external call from *C-IL* to *MASM*, a new *MASM* context is created and initialized according to the calling convention. The currently active *C-IL* context is retired to the list of inactive contexts. Constraints on the non-deterministically-chosen new context and the callee-save registers expected by the now inactive *C-IL*-context are captured in the predicate $\mathit{CIL2MASM}_{\mathit{ctxt}}(c, f, E, \mathit{regs}_{\textbf{callee}}, \mathit{context}_{\mathit{new}})$:

$$\begin{aligned} \mathit{context}_{\mathit{new}}.s &= [\mathit{frame}_{\mathit{new}}] \\ \mathit{context}_{\mathit{new}}.\mathit{regs}(R_{p_i}) &= [\![E[i-1]]\!]_c^{\theta,\pi} \quad \text{if } 1 \leq i \leq 4 \ \wedge \ i \leq \mathit{npar} \end{aligned}$$

where *npar* is a shorthand that denotes $\pi.\pi_{\mathit{MASM}}(f).\mathit{npar}$ and $[\![e]\!]_c^{\theta,\pi} \overset{def}{=} [e]^{\theta,\pi.\pi_{\mathit{C\text{-}IL}}}_{(c.\mathcal{M},c.ac)}$. The stack of the new active *MASM*-context consists of a single frame $\mathit{frame}_{\mathit{new}}$:

$$\mathit{frame}_{\mathit{new}}.p = f, \qquad \mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \qquad \mathit{frame}_{\mathit{new}}.\mathit{lifo} = []$$

$$\mathit{frame}_{\mathit{new}}.\mathit{pars}[i] = [\![E[i]]\!]_c^{\theta,\pi}, \qquad 4 \leq i < \mathit{npar}$$

Register content is chosen nondeterministically except for the values of the registers $R_{p_1}, \dots, R_{p_4}$ (parameters passed in registers according to the calling convention). The remaining parameters are passed on the stack. As a final constraint, the callee-save registers expected by the retired *C-IL*-execution-context are the same as in the new *MASM*-execution-context:

$$regs_{\mathbf{callee}} = context_{new}.regs|_{\mathcal{R}_{\mathbf{callee}} \setminus \pi.\pi_{MASM}(f).uses}$$

Note: since the *MASM*-compiler guarantees that registers in the uses list will be stored and restored properly, we only have to consider the remaining callee-save registers.

Return from MASM to C-IL. When returning, callee-save registers must have the values expected by the *C-IL*-context we return to. The content of the return-value register R_{rv} is written to the return destination *rds* given in the *C-IL*-context we return to.

Call from MASM to C-IL. Calling from *MASM* to *C-IL*, we create a new active *C-IL*-execution-context and transfer the currently active execution context to the list of inactive contexts. $MASM2CIL_{ctxt}(c, p, lifo, context_{new})$ enforces the following constraints:

$$context_{new}.s = [frame_{new}]$$

where

$$frame_{new}.f = p, \quad frame_{new}.loc = 0, \quad frame_{new}.rds = \bot$$

$$\forall npar \le i < \mathbf{len}(\mathcal{V}) : \quad \mathbf{len}(frame_{new}.\mathcal{M}_{\mathcal{E}}(v_i)) = size(t_i)$$

$$\forall 0 \le i \le 3 : (i+1) \le npar \Rightarrow read_{\mathcal{E}}^{\theta}(frame_{new}, v_i, 0, t_i) = bytes2val(regs(R_{p_{i+1}}), t_i)$$

$$\forall 4 \le i < npar : read_{\mathcal{E}}^{\theta}(frame_{new}, v_i, 0, t_i) = bytes2val(lifo[\mathbf{len}(lifo) - 1 - i], t_i)$$

The first four parameters are taken from registers, we convert their values to *C-IL*-values of the type expected by the function. The remaining parameters are passed on the stack (*lifo*) in right-to-left order.

Return from C-IL to MASM. Callee-save registers of the restored *MASM*-context stay the same, the remaining registers get assigned nondeterministically, except for R_{rv}. We convert the result of evaluation of the return expression $[\![e]\!]_c^{\theta,\pi}$ to its byte-representation and assign this value to the return value register R_{rv}.

5 Pervasive Theory

To gain a correctness result over the whole system consisting of hard- and software, we apply pervasive verification. In the pervasive stack sketched in Figure 1, adjacent models are always connected by means of simulation theorems in such a way that the abstraction provided by a higher-up layer is sound with respect to the corresponding lower layer. Our reduction theorems involve placing additional assumptions on the lower layer's execution that allow us to construct a more abstract upper layer. We only use assumptions that we can explicitly guarantee for the code we consider.

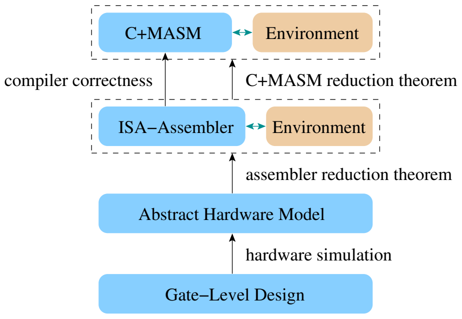


Fig. 1. Model stack for operating systems and hypervisor verification

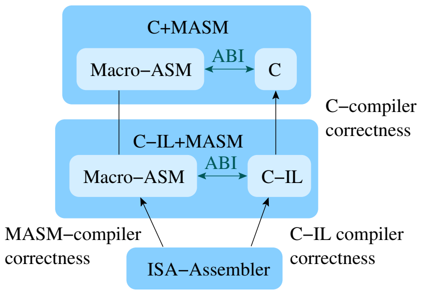


Fig. 2. Close-up view of the model stack

We are interested in justifying that our combined *C-IL+MASM*-semantics indeed correctly captures the behavior of the underlying *ISA-Assembler*-execution. In order to prove this, we make use of compiler correctness for *C-IL* and *MASM*. In particular, we rely on an explicit simulation relation that connects to the underlying *ISA-Assembler*-model (we use the term *compiler consistency relation* to refer to it).

For pure steps, we can simply apply the assumed compiler correctness. Only for inter-language steps, we need to prove that, given a state in which the compiler simulation holds for the current language, the corresponding compiler consistency relation is established. That is, after executing the compiled code of the external call, we reach an *ISA-Assembler*-configuration which is consistent with the the configuration of the abstract machine that has performed the call.

5.1 Compiler Consistency

In the actual formal definitions of the relations described in the following, we applied earlier results from the Verisoft project [6].

Memory & Code Consistency. We consider an *ISA-Assembler*-memory and a *C-IL+MASM*-memory to be consistent iff they carry identical values on all addresses except for those from the stack- or the code-region. The code region contains the compiled code of the program.

***MASM*: Register Consistency.** Configurations are consistent iff all registers of the active *MASM*-execution-context except those we abstracted away (instruction pointer, stack pointers) are the same in the *ISA-Assembler*-configuration.

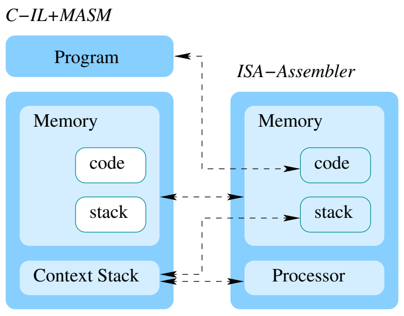


Fig. 3. Simulation of *C-IL+MASM* by the underlying *ISA-Assembler*-model

***MASM*: Stack Consistency.** Stack consistency describes on the one hand how the abstract stack matches the stack region in an *ISA-Assembler*-configuration, i.e., for the stacks of the *MASM*-contexts, how *pars*, *saved*, and *lifo* are laid out in memory and pointed to by the stack pointer registers. On the other hand, it also relates the function name and location pairs found in the abstract stack frames to the instruction pointer register, respectively the return address fields in the concrete stack layout.

***C-IL*: Stack Consistency.** For the *C-IL*-part, this describes how local memories $\mathcal{M}_{\mathcal{E}}$ from *C-IL*'s stack frames are represented in the concrete stack layout and in the processor registers (parameters in registers, register allocation for local variables). The base address of the return destination *rds* is part of the frame header. As in the *MASM*-case, control consistency is expressed over the function name/location pairs occurring in it.

Software Conditions

No Explicit Writes to Stack and Code Region. Since we explicitly manage a stack abstraction, bypassing this and writing directly to the memory region occupied by the stack will break stack consistency. Also, we do not consider semantics of self-modifying code, so the code region shall never be written. To use the described semantics, this property must be guaranteed (e.g. by performing formal verification).

No Explicit Update of Stack Pointers. For maintaining a consistent configuration (w.r.t. to the assembler execution of the compiled code) for these semantics, we should never explicitly update the stack pointers. All changes to them currently happen automatically, as a part of the $push$, pop, $call$ and ret execution of *MASM*.

6 Results

The *C-IL+MASM*-model described here has been applied to extend the baby hypervisor verification results obtained earlier [14]. In those results, there was merely specification of the effect of assembler code for the context switch between guest and hypervisor – with the presented theory, this gap has been closed. Compilation rules from *MASM* to *ISA-Assembler* and compiler consistency relations for both *C-IL* and *MASM* have been specified in full detail for the reduced version of the VAMP-processor used in baby hypervisor verification.

7 Future Work

The ideas presented can be applied such that the resulting semantics can serve as a basis for soundness proofs of translation-based assembler verification approaches, as the one described and implemented by Stefan Maus in the Vx86-tool [13].

In order to have a model on which threading libraries can be verified (including the assembler portion that actually performs the stack switch), the presented theory can be be extended to allow stack pointer updates. This work is currently in progress.

For multi-core machines, we need to consider store-buffers. Integrating the results of Cohen and Schirmer on store-buffer reduction [15] appears to be a useful step.

In order to prove correct execution of compiled code in a multi-core context, we need to place some restrictions on memory accesses in order to justify that interleaving instructions on *C-IL+MASM*-level is actually sound with respect to the underlying execution model. This work is currently in progress using an explicit ownership-model.

In an operating system and hypervisor verification effort, interrupts cannot be neglected. There is work in progress to extend the pervasive theory in such a way that interrupt handlers can be seen as additional threads interleaved with regular execution.

8 Conclusion

We have presented an integrated semantics of a simple *C*-intermediate-language and a high-level assembler language. Choosing identical memory models and stack abstraction in form of lists of stack frames makes this integration very simple. Distinguishing formally between active and inactive execution contexts, we are able to precisely model the calling conventions between *C-IL* and *MASM*. Based on earlier results, we specified compiler consistency relations for *C-IL* and *MASM* to justify that the integrated semantics presented is a sound abstraction of execution of the compiled code.

References

1. Klein, G., et al.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM, Big Sky (2009)
2. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL 2007: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 97–108. ACM, New York (2007)
3. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL 2006: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 320–333. ACM, New York (2006)
4. Ni, Z., Yu, D., Shao, Z.: Using XCAP to Certify Realistic Systems Code: Machine Context Management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
5. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. J. Autom. Reasoning 42(2-4), 301–347 (2009)
6. Leinenbach, D., Petrova, E.: Pervasive compiler verification – from verified programs to verified systems. In: 3rd Intl Workshop on Systems Software Verification (SSV 2008). Elsevier Science B. V. (2008)
7. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning 43(3), 263–288 (2009)
8. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
9. Appel, A.W.: Verified Software Toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011)
10. Morrisett, J.G., Crary, K., Glew, N., Walker, D.: Stack-Based Typed Assembly Language. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 28–52. Springer, Heidelberg (1998)

11. Gurevich, Y., Huggins, J.K.: The Semantics of the C Programming Language. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) CSL 1992. LNCS, vol. 702, pp. 274–308. Springer, Heidelberg (1993)
12. Papaspyrou, N.S.: A formal semantics for the C programming language. tech. report (1998)
13. Maus, S., Moskał, M., Schulte, W.: Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving. In: Bevilacqua, V., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 284–298. Springer, Heidelberg (2008)
14. Alkassar, E., Hillebrand, M.A., Paul, W., Petrova, E.: Automated Verification of a Small Hypervisor. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010)
15. Cohen, E., Schirmer, N.: A better reduction theorem for store buffers. CoRR abs/0909.4637 (2009)

The Location Linking Concept: A Basis for Verification of Code Using Pointers

Gregory Kulczycki[1], Hampton Smith[2], Heather Harton[2], Murali Sitaraman[2], William F. Ogden[3], and Joseph E. Hollingsworth[4]

[1] Battelle Memorial Institute,
2111 Wilson Blvd, Arlington, VA 22201, USA
kulczyckig@battelle.org

[2] School of Computing, Clemson University, Clemson, SC 29634, USA
{hamptos,hkeown,msitara}@clemson.edu

[3] Department of Computer and Information Science,
Ohio State University, Columbus, OH, 43210, USA
ogden@cse.ohio-state.edu

[4] Department of Computer Science,
Indiana University Southeast, New Albany, IN 47150, USA
jholly@ius.edu

Abstract. While the use of pointers can be minimized by language mechanisms for data abstraction, alias avoidance and control, and disciplined software development techniques, ultimately, any verifying compiler effort must be able to verify code that makes use of them. Additionally, in order to scale, the verification machinery of such a compiler must use specifications to reason about components. This paper follows a natural question that arises from combining these ideas: can the general machinery of specification-based component verification also be used to verify code that uses instances of types that are more traditionally built-in, such as arrays and pointers? This paper answers the question in the affirmative by presenting a Location_Linking_Template, a concept that captures pointer behavior, and uses it to verify the code of a simple data abstraction realized using pointers. In this deployment, pointers have a specification like any other component. We also note that the concept can be extended and realized so that different systems can plug in alternative implementations to give programmers the flexibility to choose, e.g., manual memory management or automatic garbage collection depending on their performance concerns.

Keywords: Formal specification, linked data structures, memory management, reusable components, verification.

1 Introduction

Software components must be used strictly on the basis of their specifications[17]. This is necessary for clients to understand and reason about components without concern for how they might be implemented. Implementation-neutral specifications give implementers the flexibility to provide alternative implementations for components based on different performance profiles. To verify component-based software in a scalable fashion, the verification machinery of a verifying compiler should use only the *specifications* of subcomponents[9,24,26]. We have developed and experimented with such a compiler with RESOLVE[21].

Data types commonly built into languages can also be viewed as components reused by nearly all software systems. If components are client-oriented software[1], then the software elements most frequently used by clients—built-in data types—are the quintessential components. A natural research question, therefore, is if it is possible to use the same verification machinery for verifying component-based software to also verify code that is based on built-in types such as arrays and pointers. In answering this question, this paper presents a concept for "location linking" to capture the behavior of the most complex and controversial of all data structures—the pointer.

Pointers break encapsulation [18], complicating reasoning about software. Hoare compares them to goto statements because they can "be used to create wide interfaces between parts of a program which appear to be disjoint" [7]. Despite having been well understood for over a decade, this fundamental problem remains[5] and many schemes continue to be suggested to curtail, remove, or otherwise manage the aliasing problem. Examples include ownership systems[1] and dynamic frames[10]. One well-known approach is separation logic[19], wherein procedures may define properties that hold on certain parts of the heap over their lifetime. This technique is particularly effective for links contained entirely within a component.

The approach we take for unifying verification of pointer-based code with verification of other software relies on having a formal specification of a pointer component. This idea is illustrated in Figure 1, which shows a design time diagram of a software system. In the figure, circles represent component specifications and rectangles represent realized implementations. Each implementation may depend on several other components. As an example, to reason that the Queue-Based realization is correct with respect to the specified functional behavior in the Messenger component specification, it is sufficient to know the component specifications Array and Queue. Thus, the Queue component may be implemented as an Array-Based realization or a Pointer-Based realization, but this is irrelevant to the person reasoning about the implementation of Messenger. The verification of the Pointer-Based realization in turn depends only on the specification of a pointer component. This specification-based reasoning allows components that have the same functional specification—but potentially different performance behavior—to be substituted for one another in the same system without requiring the programmer to reanalyze the entire system. This is important for maintainability and scalability.

Just as a concept exists that defines the behavior of a Queue independent from implementation, pointer behavior can also be captured in a concept. Doing so permits the system to use the same general verification machinery on pointers as on queues and other components, but does not preclude a language designer

[1] Attributed to Christine Mingins in [17].

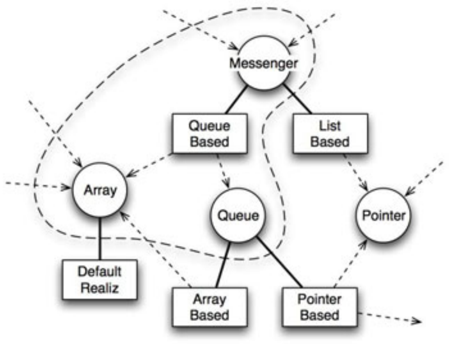


Fig. 1. Design-time diagram with component specifications and implementations

from providing syntactic sugar for pointer operations or instructing a compiler to translate such syntax as more straightforward pointer operations.

In this paper, we present specifications and code in the RESOLVE language[13,20,22], an integrated specification and programming language designed to facilitate full program verification. Though any specification language can be used to describe the components behavior, using the component in the context of the RESOLVE system highlights some of its benefits. In particular, the RESOLVE reasoning system supports a clean, value-based semantics[12] in which (1) the state space is made up of the currently defined variables and their values, and (2) the effects of a procedure call are restricted to the arguments to the call and global variables listed in the *updates* clause of the operation declaration. In addition, the RESOLVE programming language avoids unintentional aliasing from reference assignment and parameter passing, relying on swapping as its primary means of data assignment [3]. These design choices ensure that any aliasing is enacted exclusively by the pointer component. Our experience indicates that the ideas of encapsulating pointers within a component interface and avoiding most routine aliasing through swapping is applicable to, at least, C++ [8], though no formal reasoning has been attempted.

2 Specification of a System of Linked Locations

This section introduces an informal idiom for a pointer component based on a metaphor of linked locations. Figure 2 shows a diagram of an example system where symbols (Greek letters) are the information and each location has exactly one link. The eight locations to the left of the dotted line are free; the six locations to the right are occupied. The location with the slash through it is the Void location and it can never be taken. The information in free locations is always an initial value, and the links of free locations always point to Void. In the diagram, we omit these details in the interest reducing complexity.

Before a system of linked locations can be used, it has to be instantiated with an information type and a number of links. The depiction above is a result of

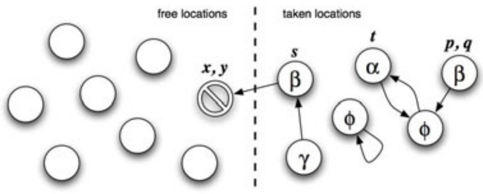


Fig. 2. A system of linked locations with symbol information and one link per location

the following instantiation, which creates a system of linked locations containing `Symbol`s, each linking to one other location:

```
Facility Symbol_Pointer_Fac is
        Location_Linking_Template(Symbol, 1)
    realized by Default_Realiz;
```

To create locations with k links, suitable for a k-ary tree representation, one would use k as the argument to instantiate the facility, instead of 1. Once instantiated, variables for working with such nodes can be declared and used as shown below.

```
Var p, q: Symbol_Pointer_Fac.Position;
```

The effects of various operations on `Position` type variables are illustrated in Figure 3. Whereas `Location` is just an abstract mathematical set with a lower bound on its cardinality, `Position` is a programming type. Conceptually, variables of this type (e.g., `p`, `q`, in the figure) each have a `Location` as their value. `Info` type values are denoted by squares in the figure.

This abstraction, of a system of linked locations, is one that serves as the most suitable basis for constructing common linked data structures. Nothing would prevent one from designing a lower-level abstraction in which pointers are modeled as $\mathbb{N}$s instead of abstract locations and providing operations for pointer arithmetic. In this case, reasoning would still require no specialized machinery, but resulting proof obligations would be more complex. In addition, by devising higher-level abstractions—for example, in which Void is ultimately reachable from all locations—implementations and reasoning of classical data structures, such as stacks, lists, and trees may be simplified. The specification machinery presented in the next section allows us to explore these trade-offs.

3 A Formal Specification

This section describes a formal specification of the Location_Linking_Template. The relationships between the mathematical objects in the concept and the notions introduced informally are straightforward. The complete specification is available in a technical report [14].

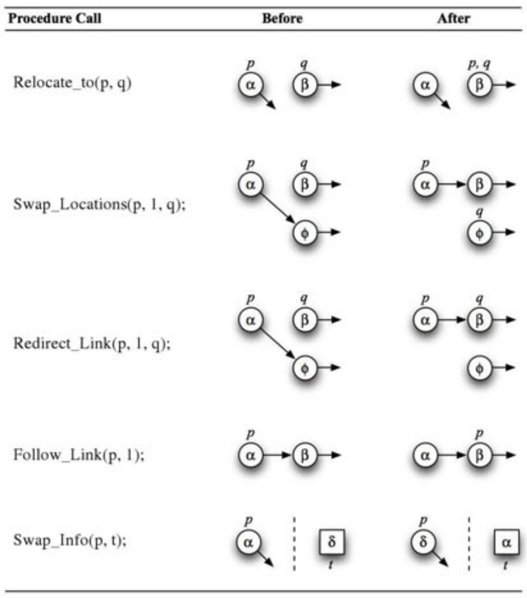


Fig. 3. The effect of selected actions on a system

Listing 1.1. A formal specification of Location_Linking_Template

```
Concept Location_Linking_Template(type Info;
        evaluates k: Integer);
    uses Function_Theory, Closure_Op_Ext;
    requires 1 ≤ k;

    Defines Location: Set;
    Defines Void: Location;
    Defines Ocpn_Disp_Incr: ℕ>0;

    Var Ref: Location×[1..k]→Location;
    Var Content: Location→Info;
    Var Occupied_Loc: ℘(Location);

    Constraints
        Info.Is_Initial[Content[Location ∼ Occupied_Loc]] ⊆
            {true} and
        Ref[(Location ∼ Occupied_Loc)×[1..k]] ⊆ {Void} and
        Void ∉ Occupied_Loc and
        ||Location|| > Total_Mem_Cap / Ocpn_Disp_Incr;

    Initialization
        ensures Occupied_Loc = ϕ;
```

```
Family Position ⊆ Location;
    exemplar p;
    initialization ensures p = Void;

    Definition Var Acessible_Loc: $\wp$(Location) =
            ({Void} ∪
             Closure_for(Location,
                 $\bigcup_{i:[1..k]}$ {λu: Location.(Ref(i, u))},
                 Position.Val_in[Position.Receptacle]);

    finalization updates Accessible_Loc;

Operation Take_New_Loc(updates p: Position);
    updates Occupied_Loc, Accessible_Loc;
    requires p ∉ Occupied_Loc and
        Ocpn_Disp_Incr + Info.Init_Disp ≤ Rem_Mem_Cap;
    ensures p ∉ #Accessible_Loc and
        Occupied_Loc = #Occupied_Loc ∪ {p};

Operation Follow_Link(updates p: Position;
        evaluates i: Integer);
    updates Accessible_Loc;
    requires p ∈ Occupied_Loc and 1 ≤ i ≤ k
        which_entails i: [1..k];
    ensures p = Ref(#p, i);

Operation Swap_Info(preserves p: Position;
        updates I: Info);
    updates Content;
    requires p ∈ Occupied_Loc;
    ensures I = #Content(p) and
        Content = λ q: Location.($\begin{cases} \#I & \text{if } q = p \\ \#Content(q) & \text{otherwise} \end{cases}$);

(* Other operations omitted for brevity. *)

end;
```

The hash sign (#) indicates the value of a variable before a call. The set $\mathbb{N}^{>0}$ are those natural numbers above zero (i.e., all the natural numbers save zero). We use S ~ R as our notation for the set difference of S and R.

3.1 Shared Conceptual State

The shared conceptual state of `Location_Linking_Template` is specified through *defines* clauses, conceptual (or "specification") variable declarations, and their constraints. The *defines* clauses at the beginning are placeholders for deferred implementation-dependent definitions. Though we expect that objects of type

Location will somehow be tied to a machine's memory addresses, for the purposes of reasoning about this component the programmer need only know that `Location` is a set, `Void` is a specific location, and `Ocpn_Disp_Incr` is a non-zero natural number that represents the memory overhead for a `Location`. `Total_Memory_Capacity` is a global variable across the system.

Objects of type `Location` correspond to the notion of locations described in Section 2. The type parameter, `Info`, indicates the type of information that a location contains, while the second parameter, `k`, indicates the number of links from a given location. The three conceptual variables near the beginning of the concept enable clean mathematical reasoning about locations: `Ref(q, i)` returns the location targeted by the i^{th} link of `q`, `Content(q)` returns the information at a given location `q`, and `Occupied_Loc` is the set of all non-free locations.

The first conjunct of the *constraints* clause asserts that all the unoccupied locations have initial information value. It uses the square bracket notation to lift a function from one that operation on a domain `D` and a range `R`, to one that operates on the powerset of `D`, unioning all results, and returning a value in the powerset of `R`. So the first conjunct unions the result of testing to see if each `Info` in unoccupied locations is an initial value—which must be the singleton set containing only `true`, i.e., all unoccupied locations must always contain initial values. The second conjunct ensures that each link of each unoccupied location references `Void`. The third ensures that `Void` cannot become an occupied location. And the last conjunct ensures that the `Location` set is at least as big as necessary.

The assertion that free locations have default information and default links exists strictly for reasoning about functional behavior. The performance part of this specification assumes that no memory is allocated for information until the `Take_New_Loc` operation is invoked. The value of `Ocpn_Disp_Incr` is the overhead space that a newly taken location with `k` links occupies in memory.

Once a system of locations is instantiated, the *initialization* clause ensures that all locations in the newly created system are unoccupied, and the constraints clause ensures that all of these free locations have default information and default links. The default target for links is the `Void` location.

3.2 Position Type

The *initialization ensures* clause asserts that `p = Void`. Since the symbol `p` that occurs here is the mathematical value of the programming variable `p` rather than the programming variable p itself, the assertion is interpreted as "The location of the variable named `p` is `Void`." The clause indicates that all new pointer variables are conceptually at the `Void` location.

The *exemplar* clause simply introduces an example position variable so that the name can be used in the scope of the type declaration.

The mathematical definition `Accesible_Loc` is a variable because its value depends not only on the value of its parameter, `q`, but also the conceptual variables `Ref` and `Occupied_Loc`; so the same location may be accessible in one program state and inaccessible in the next. Variable definitions such as this are simply a notational convenience, as the necessary conceptual variables can be passed explicitly as parameters. `Accesible_Loc` states that a Location is accessible if it is `Void` or in the closure of all links from all Locations starting from any named position variable (i.e., a “receptacle”) currently in use.

`Receptacle` is a meta variable built into the specification language and it represents the set of all variables of a given type (in this case, `Position`) currently in scope, while `Val_in()` is a built-in function that takes an element of some type’s `Receptacle` set and returns its corresponding mathematical value (in this case, a `Location`). When a variable is declared of type `Position`, its unique identifier is added to `Position.Receptacle`. When a variable is destroyed (goes out of scope) the identifier is removed.

When an operation can potentially modify state that affects a definition variable such as `Accessible_Loc`, we include the variable in the updates clause; ones not specified to be updated are not affected, providing an implicit frame property. For example, the finalization (which acts as an implicit operation) includes `Accessible_Loc` in its *updates* clauses, since the destruction of a `Position` may impact the set of accessible locations if one of its links referenced a location that was otherwise inaccessible.

Note that, because of RESOLVE’s value semantics, an operation cannot affect a variable outside it’s available scope save through a well-specified interaction such as those provided by the component introduced in this paper. In that case, it would be the responsibility of the components sharing `Positions` to specify how their shared state is permitted to change.

3.3 Operations

The management actions informally described in Section 2 correspond directly to the operations given in the concept.

The `Take_New_Loc` operation takes a single position variable as a parameter. The *updates* parameter mode indicates to the client that the operation modifies the value of p. The *updates clause* on the following line gives the conceptual (state) variables that we can expect this operation to modify. In this case, we can expect the operation to affect both the occupation status of one or more locations and the accessibility of the system.

The requires clause guarantees that `p` cannot reside at a taken location and that sufficient memory exists. Since the `Void` location is perpetually free, it will be the location where p typically resides when the operation is called. In general, performance behavior such as memory use should be decoupled from the behavioral spec, but here sufficient memory is a constraint of any conceivable implementation. Performance specification is discussed further in [6,23].

The *ensures* clause guarantees that the newly taken location was not previously accessible and that the set of occupied locations is extended by precisely the newly occupied location. Because of the implicit frame property, and because `Content` and `Ref` are not mentioned in the *updates* clause, a client can be sure that the newly taken location has default information and that all of its links point to the `Void` location.

The parameters in these operations have various modes that summarize the operation's effect. The *updates* mode has already been mentioned. The *clears* mode ensures that an argument will return with an initial value. The *preserves* mode prohibits any changes to an argument's value. The *replaces* mode indicates that the incoming value will be ignored and replaced with a meaningful one. The *evaluates* mode indicates that the operation expects an expression, which modifies the parameter swapping behavior.

We show only specifications of those operations that are used in the code in the next section. Operation `Follow_Link` causes a position to point to the target of one of its links, whereas `Swap_Contents` exchanges the `Info` associated with a given location with the given `Info`. The interested reader is referred to [14] for specifications of additional operations to redirect a position variable's link, abandon one, etc.

4 Memory Management

Through its operations, `Location_Linking_Template` provides all the functionality of traditional pointers. For example, the client can obtain the benefits of aliasing by positioning two or more variables at the same location. But the concept also allows the client to fall into the traditional traps involving pointers: dangling references and memory leaks. This section looks at different ways these problems can be managed.

4.1 Performance and Extensions

A dangling reference occurs when a location is free but remains accessible, as in the following code.

```
Var x, y: Position;
Take_New_Loc(x);
Relocate(y, x);
Abandon_Location(x);
```

When `x` abandoned its location, the location's status changed from taken to free. Though `x` was relocated to `Void`, `y` remained at the location, so the location continues to be accessible. `Position` variables are effectively bound to the type of `Info` during instantiation, so there is no danger of inadvertently modifying (through the dangling reference) the contents of a memory location that is being used by another variable somewhere else in the program. Real memory locations on a machine are limited, so the specification permits implementations that can reclaim memory even if a dangling reference existed for them. The `Is_Occ` operation (provided in an extension to the concept and shown in Figure 1.2) effectively tells the client whether a variable contains a dangling reference. Since a `Position` variable resides at the location in question, the location is accessible. If the location is taken, it is usable by the client; if the location is free, the client cannot affect it.

Listing 1.2. Extensions to Location_Linking_Template

```
Extension Occ_Checking_Capability
        for Location_Linking_Template;
    Operation Is_Occ(preserves p: Position): Boolean;
        ensures Is_Occ = (p ∈ Occupied_Loc);
end;

Extension Cleanup_Capability for Location_Linking_Template;

    Operation Abandon_Useless();
        updates Occupied_Loc, Content, Ref;
        ensures Occupied_Loc =
                #Occupied_Loc ∩ Accessible_Loc and
            Content↿((Location ∼ #Occupied_Loc) ∪
                    Accessible_Loc) =
                #Content↿((Location ∼ #Occupied_Loc) ∪
                    Accessible_Loc) and
            Info.Is_Initial[
                Content[#Occupied_Loc ∼ Occupied_Loc]] =
                {true} and
            Ref = λ q: Location,
                λ j: [1..k].(
                    $\begin{cases} \text{Void} & \text{if } q \in \#\text{Occupied\_Loc} \sim \\ & \quad \text{Occupied\_Loc} \\ \#\text{Ref}(q, j) & \text{otherwise} \end{cases}$ );
end;
```

A memory leak occurs when a location is taken but not accessible. The following code segment creates a memory leak.

```
Var x, y: Position;
Take_New_Loc(x);
Relocate(x, y);
```

The location that was taken by `x` continues to have a taken status but has become inaccessible. The operation that performs garbage collection, `Abandon_Useless`, is provided in an *extension* to the concept: a kind of module that introduces functionality not all implementations will want to provide.

A garbage collecting implementation of `Location_Linking_Template` would also provide the `Abandon_Useless` operation. A client may then choose to ignore the `Abandon_Location` operation and periodically invoke `Abandon_Useless` operation. Extensions for other garbage collection strategies, such as mark-and-sweep, could also be provided.

4.2 Implementation Flexibility

A given programming language will typically hardcode the choice of implementation for the location linking concept, but the concept itself allows implementation options. If a language allows options, then a facility declaration mechansim such as in RESOLVE can be used. The following declaration creates a pointer facility containing `Queue` information with one link per location, providing no extension operations and thus leaving the burden of memory management on the client.

```
Facility Queue_Pointer_Fac is Location_Linking_Template(
        Queue, 1)
    realized by Default_Realiz;
```

A garbage collecting implementation would additionally implement the `Cleanup_Capability` extension:

```
Facility GC_Queue_Pointer_Fac is
        Location_Linking_Template(Queue, 1)
    extended by Cleanup_Capability
        realized by Garbage_Collecting_Realiz;
```

Each facility acts as a distinct component. Thus, an object of type `Queue_Pointer_Fac.Position` cannot be used where an object of type `GC_Queue_Pointer_Fac.Position` is expected.

The stack component described in the next section contains a local pointer facility and uses a form of manual component-level memory management[16]. Manual memory management for global pointer facilities becomes more difficult with each realization that imports the facility, because all realizations that import the same facility share the same set of locations. Therefore, global pointer facilities may be good candidates for garbage collection implementations. The facility mechanism allows multiple pointer components with different implementations to exist in the same program.

5 Application

Using `Location_Linking_Template` to implement linked data structures will be familiar to anyone who has implemented a linked list in a language with explicit pointers such as C or Pascal, though the nomenclature is different. This section gives excerpts from a stack data abstraction implemented using `Location_Linking_Template`, along with the necessary verification conditions (VCs) generated by the RESOLVE verifying compiler in establishing its correctness. The key point is that the compiler uses the same verification machinery for generation of these VCs as for any other code based on a formal specification. Our VC-generation mechanism is established in [4].

Due to space constraints, we make a number of simplifying assumptions. We assume that an unlimited number of locations exist, so the notions of occupied locations or abandoning locations are not used. The `Stack` specification is *unbounded*. Similarly, since a stack implementation requires only one link, we fix k at 1 so that, for example, the `Follow_Link` operation does not have an argument that indicates which link to follow.

Because our compiler accepts inputs in ASCII, we use ASCII notations of mathematical concepts here.

5.1 Specification of a Stack Concept

Just as a Pointer is modeled mathematically as a Location containing a generic type, a Stack is modeled mathematically as a string (i.e., a finite sequence) of genericly-typed entries.

```
Concept Unbounded_Stack_Template(type Entry);
        uses String_Theory;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        initialization ensures S = empty_string;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires S /= empty_string;
        ensures #S = <R> o S;

    (* Other operations omitted *)

end Unbounded_Stack_Template;
```

5.2 Pointer-Based Implementation of Stacks

In this implementation, the `Stack` type is represented by a `Position`. This requires an instantiation of `Location_Linking_Template_1`. The representation convention (or invariant) uses a locally-defined predicate `Is_Void_Reachable` that is true **iff** `Void` can be reached by following links defined by some reference function (like, for example, `Ref` from `Location_Linking_Template_1`). The present implementation does not share locations among different stacks, so a corresponding invariant (not shown) is necessary. It is also possible to develop an implementation where stacks share locations. The correspondence (or abstraction) function uses another definition, `Str_Info`, that takes a `Location`, as well as a content function and a linking function, and returns the sequence of `Info` elements contained along its link chain as a string[2].

```
Realization Location_Linking_Realization for Stack_Template;
        uses Location_Linking_Template;

    Facility Entry_Ptr_Fac is
            Location_Linking_Template_1(Entry)
        realized by Std_Location_Linking_One_Realiz;

    Type Stack is represented by Entry_Ptr_Fac.Position;
        convention Is_Void_Reachable(S, Entry_Ptr_Fac.Ref);
            (* Locations-not-shared invariant omitted *)
        correspondence Conc.S = Str_Info(S,
            Entry_Ptr_Fac.Content, Entry_Ptr_Fac.Ref);
```

[2] The actual definitions of `Str_Info` and `Is_Void_Reachable` are omitted for brevity.

```
    Procedure Pop(replaces R: Entry; updates S: Stack);
        Swap_Contents(S, R);
        Follow_Link(S);
    end;

    (* Other Procedures omitted *)
end;
```

Note that the facility's location pool is local and represents an encapsulated, private heap that is inaccessible outside this family of Stacks. This simplifies reasoning significantly by providing an implicit frame property. One could define a heap in a global facility, instead, though this would complicate reasoning.

5.3 Verification Process

Applying the specification-based verification machinery yields the VCs found in Table 1, which arise from the ensures clauses of Stack operations, requires clauses of called location linking operations, and showing conventions hold at the end of each procedure. Proofs of the VCs are straightforward and can be handled by most provers, such as those summarized in [11].

As an example, consider the following VC for establishing the convention at the end of a call to Pop[3]:

```
Goal:
Is_Void_Reachable(Ref(S), Ref)

Given:
Is_Void_Reachable(S, Ref) and
Str_Info(S, Content, Ref) /= empty_string
```

Note that this is a straightforward proof because the prover employs pre-established theorems about reachability so that we know the second given is true **iff** S /= Void. Given that, if S's links can be followed to Void given the reference function Ref, then something that S links to directly can also be followed to Void under the same reference function.

5.4 Closely Related Work

The closure results necessary for proofs such as reachability are established independently in the math module Closure_Op_Ext (seen imported in Listing 1.1.) Such factoring out of reusable mathematical development (independent of their application to the present verification problem) is a key reason for the simplicity of this treatment, compared to, for example, [15].

An important direction for further research is experimentation with an assortment of benchmarks, such as the ones discussed in [2]. However, we hypothesize that the assertions and complexity of their proofs are likely to be different

[3] Irrelevant conjuncts of the "Given:" portion have been removed for brevity.

Table 1. VCs for Location_Linking_Realization

VC	Given	Goal
1	true	Is_Void_Reachable(Void, Ref)
2	true	Str_Info(Void, Content, Ref) = empty_string
3	Str_Info(S, Content, Ref) /= empty_string	S /= Void
4	Str_Info(S, Content, Ref) /= empty_string	S /= Void
5	Is_Void_Reachable(S, Ref)	Is_Void_Reachable(Ref(S), Ref)
6	Is_Void_Reachable(S, Ref)	Str_Info(S, Content, Ref) = (<Content(S)> o Str_Info(Ref(S), lambda L: Z ({R if L = S; Content(L) otherwise}), Ref))
7	true	Void = Void
8	Temp' /= Void	Temp' /= Void
9	Temp' /= Void	Temp' /= Void
10	Is_Void_Reachable(S, Ref)	Is_Void_Reachable(Temp', lambda L: Z ({S if L = Temp'; Ref(L) otherwise}))
11	Is_Void_Reachable(S, Ref)	Is_Void_Reachable(S, Ref)
12	true	(S = Void) = (Str_Info(S, Content, Ref) = empty_string)
13	true	Str_Info(S, Content, Ref) = Str_Info(S, Content, Ref)

from those resulting from the approach discussed in this paper because of language design differences and our use of pre-verified components. For example, an always-void-reaching concept, such as the one in Section 2, would lead to much simpler invariants for list implementations.

It will also be interesting to study our approach to tree structures in relation to [25]. Our approach similarly involves establishing a mathematical theory of tree structures and using it to specify and reason about a `Tree` concept. However, the pointer-based implementation of the concept will be hidden (and verified once) using the ideas in this paper. Thus, such details will not routinely be raised in verification of client code. Given these simplifications, it may be interesting to adapt the decision procedure presented there and determine if additional theoretical or performance gains can be achieved in such a setting.

6 Summary

We have presented a formal specification of a concept to capture pointer behavior. The specification is designed such that extensions to the basic specification can give language designers and programmers the flexibility to choose between manual memory management and automatic garbage collection based on their performance concerns. We have shown that a verifying compiler with the necessary machinery to reason about component-based software via the specifications of reusable components can be used naturally to verify pointer-based code using the given specification.

Acknowledgements. This research is funded by NSF grants CCF-0811748 and DMS-0701187. We would like to thank all current and past members of the RESOLVE Software Research Group at The Ohio State University and Clemson University, in particular Ray McDowell, whose comments have helped improve the specifications. Our thanks are also due to the anonymous referees for their extensive comments.

References

1. Banerjee, A., Naumann, D.A.: State Based Ownership, Reentrance, and Encapsulation. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 387–411. Springer, Heidelberg (2005)
2. Böhme, S., Moskal, M.: Heaps and Data Structures: A Challenge for Automated Provers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 177–191. Springer, Heidelberg (2011), `http://dx.doi.org/10.1007/978-3-642-22438-6_15`
3. Harms, D.E., Weide, B.W.: Copying and swapping: Influences on the design of reusable software components. IEEE Trans. Softw. Eng. 17, 424–435 (1991), `http://dl.acm.org/citation.cfm?id=114769.114773`
4. Harton, H.: Mechanical and Modular Verification Condition Generation for Object-Based Software. Phd dissertation. Clemson University, School of Computing (December 2011)
5. Hatcliff, J., Leavens, G.T., Rustan, K., Leino, M., Müller, P., Parkinson, M.: Behavioral interface specification languages (2009), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.150.723`
6. Hehner, E.C.R.: Formalization of time and space. Formal Aspects of Computing 10, 290–306 (1998), `http://dx.doi.org/10.1007/s001650050017`
7. Hoare, C.A.R.: Recursive data structures. In: Hoare, C.A.R., Jones, C.B. (eds.) Essays in Computing Science. Prentice-Hall, New York (1989)
8. Hollingsworth, J.E., Blankenship, L., Weide, B.W.: Experience report: using RESOLVE/C++ for commercial software. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications, SIGSOFT 2000/FSE-8, pp. 11–19. ACM, New York (2000), `http://doi.acm.org/10.1145/355045.355048`
9. Jones, C.B.: Systematic software development using VDM. Prentice Hall International (UK) Ltd., Hertfordshire (1986)
10. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
11. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M.A., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience Report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)

12. Kulczycki, G.: Direct Reasoning. Phd dissertation. Clemson University, School of Computing (January 2004)
13. Kulczycki, G., Sitaraman, M., Roche, K., Yasmin, N.: Formal specification. In: Wah, B.W. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc. (2008)
14. Kulczycki, G., Smith, H., Harton, H., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: Technical report RSRG-11-04, The Location Linking Concept: A Basis for Verification of Code Using Pointers (September 2011), http://www.cs.clemson.edu/group/resolve/reports.html
15. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 171–182. ACM, New York (2008), http://doi.acm.org/10.1145/1328438.1328461
16. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc., Upper Saddle River (1988)
17. Meyer, B.: On to components. Computer 32, 139–140 (1999)
18. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
19. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, pp. 55–74. IEEE Computer Society, Washington, DC, USA (2002), http://dl.acm.org/citation.cfm?id=645683.664578
20. Sitaraman, M., Weide, B.: Component-based software using resolve. SIGSOFT Softw. Eng. Notes 19, 21–22 (1994), http://doi.acm.org/10.1145/190679.199221
21. Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H., Harton, H., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.: Building a push-button resolve verifier: Progress and challenges. Formal Aspects of Computing 23, 607–626 (2011), http://dx.doi.org/10.1007/s00165-010-0154-3
22. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about Software-Component Behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
23. Sitaraman, M., Kulczycki, G., Krone, J., Ogden, W.F., Reddy, A.L.N.: Performance specification of software components. In: SSR, pp. 3–10 (2001)
24. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River (1989)
25. Wies, T., Muñiz, M., Kuncak, V.: An Efficient Decision Procedure for Imperative Tree Data Structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032266.2032302
26. Wing, J.M.: A specifier's introduction to formal methods. Computer 23, 8–23 (1990), http://dl.acm.org/citation.cfm?id=102815.102816

Verifying Implementations of Security Protocols by Refinement

Nadia Polikarpova[1] and Michał Moskal[2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
nadia.polikapova@inf.ethz.ch
[2] Microsoft Research Redmond
michal.moskal@microsoft.com

Abstract. We propose a technique for verifying high-level security properties of cryptographic protocol implementations based on stepwise refinement. Our refinement strategy supports reasoning about abstract protocol descriptions in the symbolic model of cryptography and gradually concretizing them towards executable code. We have implemented the technique within a general-purpose program verifier VCC and applied it to an extract from a draft reference implementation of Trusted Platform Module, written in C.

1 Introduction

Vulnerabilities in security-critical code can arise either from defects in underlying cryptographic protocols, or from inconsistencies between the protocol and the implementation. A lot of research in security is devoted to verifying abstract protocol definitions against high-level *security goals*, such as secrecy and authentication. An example of such a goal could be that an honest client only accepts a message from a server, if it is related to a previously made request.

On the implementation side, techniques of static and dynamic program analysis are applicable for checking low-level properties of the code, such as absence of buffer overruns. A challenge that is addressed by very few existing approaches is bridging the gap between the two sides, that is, verifying that a program implements a certain protocol and accomplishes the same security goals.

Most protocol implementations are written manually in an imperative programming language, often C. In this work we use VCC [5], a general-purpose verifier for C programs, to prove the security of both the protocol and its implementation.

This task poses two main challenges. First, VCC is a verifier for arbitrary functional properties but does not support expressing secrecy and authentication in a straightforward manner. Second, it is desirable to decouple the security goals of the protocol from the properties of its implementation to allow for simpler specifications and more modular proofs.

VCC has been used previously for verification of cryptographic protocols [8]. Our work is an extension thereupon, with two main differences. First, our approach relies entirely on VCC and does not require external security proofs in Coq, which avoids translation between different formalisms and utilizes VCC strengths in handling mutable state. Second, we have applied the technique to a more complex, stateful protocol, which prompted the scalability problems and thus development of our refinement-based verification approach, which is the main contribution of this paper.

Our refinement strategy consist of three levels. The initial model (L0) describes a security protocol on the level of abstraction of a conventional protocol definition language (message sequence chart, role script and similar). The purpose of the first refinement (L1) is to switch from an abstract representation of protocol messages to their concrete byte string format. Finally, the second refinement (L2) connects the protocol model to the real implementation.

We applied this approach to a key management protocol from a slightly simplified version of the code submitted by Microsoft to the Trusted Computing Group for the reference implementation of the Trusted Platform Module version 2. The simplifications we made are described in Sect. 5; however note that we did not change the data structures that store cryptographic keys or the code manipulating those keys. We therefore believe that the case study still represents a realistic protocol implementation in C; moreover it is not a standalone program, but a part of a software system of substantial size.

The next two sections introduce the VCC verifier and the Trusted Platform Module. Sect. 4 describes the proposed refinement approach in detail. Sect. 5 summarizes the results of our case study, Sect. 6 discusses some related work and Sect. 7 concludes.

2 Background: The VCC Verifier

VCC is a deductive verifier for C programs. It checks that a program adheres to a specification in the form of inline assertions, function pre- and post-conditions and *object invariants*, which are associated with compound C types (struct and unions). It works by generating verification conditions from the program and its specification, via the Boogie intermediate language [2]. These conditions are discharged by a reasoning engine of choice, usually Z3 [6].

The specifications are provided as annotations directly in the source code and are expressed in a language that includes the C expression language, and extends it with quantification, user-defined predicates and functions. Additionally, the annotations can provide ghost code manipulating ghost data, i.e., code and data removed before the program is executed. VCC supports unbounded integers, maps between arbitrary types (defined using lambda-expressions), and user-defined algebraic data-types, which all can be used in ghost code. This added expressivity is useful for specifying data structures. For example, a red-black tree can be equipped with a ghost field of a map type representing the current value of the tree; functions operating on the tree can then be fully functionally specified with respect to that field. We shall use a very similar mechanism to represent the state of a protocol execution.

In VCC object invariants specify ownership relations: an object would usually own all the objects that comprise its internal representation. Together with ghost fields abstracting over values of complex object trees, ownership provides for data abstraction.

Our approach to verifying security properties can be applied to any verifier with a similar feature-set. While both ownership and invariants are built into VCC, we believe the technique would work equally well if they were implemented on top of, e.g., separation logic in a tool like VeriFast [10].

```
<public, private> =
    Create(parent_handle);
// other code
// ...
handle = Load(parent_handle,
    public, private);
plain = Decrypt(handle, cipher);
```

Listing 1: Example usage of a storage key

```
Create(parent_handle) {
    <public, sensitive> = Random();
    integrity = Hash(<sensitive, public>);
    private = Cipher(<integrity, sensitive>,
        protection_key(parent_handle));
    return <public, private>;
}
```

Listing 2: Pseudocode of Create

3 Case Study: Trusted Platform Module

A Trusted Platform Module (TPM) is a secure cryptoprocessor used in most modern personal computers. A TPM device is equipped with volatile and non-volatile memory and can generate cryptographic keys, restrict their further usage, perform random number generation, as well as other cryptographic operations (hashing, encryption, signing). TPM can provide a wide range of hardware security services, most popular being cryptographic key management, ensuring platform integrity and disk encryption (for example the BitLocker Drive Encryption feature of Microsoft Windows).

Being a member of Trusted Computing Group, Microsoft has submitted for comments a draft of the new generation of the TPM standard: version 2.0. It consists of about 1000 pages of natural language specification and a reference implementation in C (approximately 30000 lines of code). TPM[1] supports a total of 102 commands, such as create a cryptographic key, load a key for further use, decrypt a ciphertext, etc.

In this case study we restrict ourselves to the key management functionality of TPM, as it is the core part needed for most applications. Keys used by the TPM (also called *objects*)[2] are organized in a hierarchy, where child keys are protected by their parents. The TPM specification poses a restriction that objects protecting other objects (called *storage keys*) are not allowed to be imported, that is, have to be created and used by the same TPM. We only consider management of such storage keys.

Internally a storage key consists of a *public* and a *sensitive* area. The former contains an asymmetric public key together with object attributes and parameters; the latter consists of an asymmetric private key as well as a *symmetric protection key*. Whenever an object resides in the TPM internal memory, its sensitive area can be stored unencrypted; however when the object is stored off TPM, its sensitive area is *protected* by the protection key of its parent. A protected sensitive area is called a *private* area.

3.1 Creating and Loading Objects

A typical usage scenario of a storage key is given in Listing 1. To create a key one invokes the Create command of the TPM, which returns the public and private areas of the new object. Before the key can be used (e.g., to decrypt data or protect other keys), it needs to be loaded into the internal memory of the TPM using the Load command.

We would like to verify the following two properties about this scenario:

[1] For brevity, we will use this name to refer to the draft specification.

[2] Note that the notion of object in TPM (a cryptographic key) is different from object in VCC (an instance of a compound type), but the former can be implemented using the latter.

1. sensitive areas of all the objects in the hierarchy remain secret (not known outside the TPM);
2. whenever a storage key is successfully loaded into a TPM under some parent, it had been created by the same TPM under the same parent.

Looking at an abstract description of Create (Listing 2), it is easy to reason informally that those properties are ensured by the format of the private area. Indeed, an external client cannot encrypt arbitrary messages with a secret protection key, thus to load a fake object he would have to *guess* two values pub and $priv$, such that, if $priv$ decrypts to a pair $< i, s >$, then $i = h(s, pub)$ (where h is a known hash function).

3.2 Key Management as a Security Protocol

The scenario described above can be expressed as a single-message security protocol, where an agent C (Create) sends a message to an agent L (Load):

$$C \rightarrow L : par, p, \{|h(p, s), s|\}_{k(C,L,par)}$$

Here comma is used for pairing, $\{| \cdot |\}$ is symmetric encryption, par is an identifier of the parent, p and s are the public and the sensitive area of an object and $k(C, L, par)$ is a key shared between C and L (the protection key of the parent).

The security goals listed above can be expressed as follows:

1. (*secrecy*): s, $k(C, L, par)$ are secret, shared between agents C and L;
2. (*authentication)*: agent L agrees with C on par, p, s; that is, whenever L receives a message of the above format, he can be sure that is has once been sent by C with the same parameters par, p and s.

Symbolic formalisms for protocol verification usually consider agents communicating through a network controlled by a *Dolev-Yao attacker* [7]. Such an attacker can eavesdrop, decompose and recompose messages, but never relies on luck, e.g., he cannot guess a secret key or accidentally obtain a required hash value due to a collision. We adopt the same attacker model for our approach.

4 Refinement Approach

To deal with the complexity of specification and verification of real-world security code we propose an approach based on stepwise refinement in the style of Event-B [1]. The main idea is to first reason about a high-level protocol description (like the one given in the previous section) and then gradually concretize it towards the real implementation. Refinement enables us to separate different aspects of the problem, in particular the security of the protocol and whether the protocol is correctly implemented by the code.

In Event-B the *initial model* of a system consists of an abstract state space constrained by invariants and a set of events that update the abstract state. The purpose of the invariants is to express high-level system requirements (in our case, the security goals) in a formal but straightforward way; all events have to be proved to maintain the invariants.

```
_(ghost typedef struct {
  // Unbounded integer:
  \integer a;
  // Invariant on the state space:
  _(invariant a > 0)
} State_0;
State_0 s0;

void event_0()
  _(updates &s0)
  _(ensures s0.a == \old(s0.a) + 1)
{
  _(unwrapping &s0) {
    s0.a = s0.a + 1;
  }
})
```

Listing 3: An example initial model in VCC

```
_(ghost typedef struct {
  \integer b, c;
  _(invariant \mine(&s0)) // Ownership
  // Gluing invariant:
  _(invariant s0.a == b + c)
} State_1;
State_1 s1;

void event_1()
  _(updates &s1)
  _(ensures s1.b == \old(s1.b) + 1)
{
  _(unwrapping &s1) {
    event_0();
    s1.b = s1.b + 1;
  }
})
```

Listing 4: An example refinement in VCC

Each *refinement* introduces new (concrete) variables that are linked to the old (abstract) variables by a *gluing invariant*. An essential feature of stepwise refinement is that verifying a concrete event (that manipulates concrete variables) does not require re-verifying preservation of the abstract invariants, but only of the gluing invariants. This enables decomposition of proofs in addition to specifications.

4.1 Refinement in VCC

Unlike Event-B we do not use a special-purpose notation and tool support, but rather encode refinement within a general-purpose program verifier. One of the benefits is a seamless transition from the model to the actual implementation, without relying on a code generator.

In VCC we represent the state space of the initial model as a ghost struct and often make use of unbounded integers and infinite maps to achieve a higher level of abstraction (see Listing 3 for an example). An event is encoded as a ghost function with a contract _(**updates** &state). This contract allows the function to open up the state object (using a statement _(**unwrapping** &state)) and modify it, as long as its invariant is preserved, which is exactly the proof obligation we would like to impose on an event.

At least one of the events in a model has to ensure the invariant without requiring it (it is called the *initialization event*), in order to make sure that the invariant is consistent.

On each refinement level concrete state is encoded as a struct (physical if it is the final refinement, and ghost otherwise), equipped with a gluing invariant (Listing 4). The ownership system of VCC makes it possible to express the refinement relation in an elegant way. VCC does not allow mentioning abstract variables in the gluing invariant, unless the abstract state is *owned* by the concrete state. The system then guarantees that whenever the concrete state is known to be consistent (i.e., in between executing the events) no code could modify the abstract state without updating the concrete variables accordingly.

Concrete events are encoded as functions with an _(**updates** ...) contract for concrete state. A body of such a function calls its abstract counterpart and then updates the concrete state to reestablish the gluing invariant. Because the abstract event is already known to maintain the invariant of the abstract state, only the preservation of the gluing invariant has to be proved at this point.

Note that this approach uses refinement only as a methodology for structuring specifications and proofs; the validity of verification results does not depend on maintaining a simulation relation between the models on different levels of abstraction. Using the example above, if the postcondition of event_1 faithfully describes its intended effect, and the invariants of State_0 and State_1 capture all the system requirements, then it does not matter if event_1 refines event_0; in fact, the latter can be eliminated from the system, letting the former update s0 directly (and thus take up the burden of proving the preservation of s0's invariant).

In the next three sections we describe in detail the three models (the initial model and its two refinements) that we propose for verifying protocol implementations. We explain the approach in terms of the create-load protocol from our case study, however we believe that the description extends naturally to other similar protocols and security goals.

4.2 High-Level Protocol Definition (L0)

Our initial model describes the protocol in the formalism of symbolic cryptography. In this formalism the set of messages is modelled as a *term algebra*, which in VCC can be defined using the **datatype** construct (Listing 6).[3]

In our example the algebra consists of byte string literals, terms denoting cryptographic operations (encryption and hashing) as well as three different kinds of *compound terms*: Sensitive, Object and Private. These three constructors essentially all represent pairs; we distinguish them, as their concrete representations in the actual code have different formats. To justify the distinction, we have to prove that no two compound terms of different kinds can correspond to the same concrete representation, which is discussed in Sect. 4.3.

The variables of the initial model keep track of the state of a protocol execution, namely:

– internal TPM state: the object hierarchy and loaded objects;
– all the terms created by honest participants (i.e., the TPM) as part of the protocol messages;
– all the terms known to the attacker, either eavesdropped or constructed.

We use infinite maps (denoted **bool** ...[Term]) to represent sets of terms through their characteristic functions. The state space is encoded in a ghost struct called Log

[3] The listings in this section are somewhat simplified for clarity; the full VCC source code of our case study is available under `http://se.inf.ethz.ch/people/polikarpova/tpm.zip`

```
_(ghost
typedef struct {
  // Objects in the hierarchy
  bool objects[Term];
  // Mapping from an object to its parent
  Term parent[Term];
  // Loaded objects
  bool loaded[Term];
  // Terms generated by honest agents
  bool protocol[Term];
  // Terms known to the attacker
  bool attacker[Term];

  _(invariant \forall Term pub, sen; objects[Object(pub, sen)] ==> !attacker[sen])
  _(invariant \forall Term pub, sen, k; is_object_sym_key(\this, k) ==>
    attacker[Cipher(Private(Hash(Integrity(sen, pub)), sen), k)] ==>
      objects[Object(pub, sen)] && symkey(parent[Object(pub, sen)]) == k)
  // ... more invariants ...
} Log;
Log log;)
```

Listing 5: Protocol log

```
_(datatype Term {
  case Literal(ByteString s);
  case Sensitive(Term skey, Term symkey);
  case Object(Term pub, Term sen);
  case Private(Term int_hash, Term sen);
  case Cipher(Term t, Term k);
  case Hash(Term t);
})
```

Listing 6: Term algebra

(Listing 5). We also define a function **bool** used(Log, Term) that represents the union of protocol and attacker sets of a log.

Each event of the model encodes a protocol step of one of three kinds: either an honest agent *sending* a message (revealing it to the attacker), an honest agent *receiving* a message, or the attacker constructing a new message from his current knowledge.

Secrecy goals are encoded as invariants of the log, restricting the attacker set. The first of the object invariants in Listing 5 is an example of such a *security invariant*: it states that for all objects in the hierarchy their sensitive area is never known to the attacker. The second one is an *auxiliary invariant*, which is not part of the requirements and is only used to assist verification.

An authentication goal is always associated with an honest agent receiving a message: upon receipt the agent wants to be sure that the message came from a certain principal. Thus authentication goals are encoded as *security postconditions* of receive events (see Load for an example).

Verifying the events against the security invariants and postconditions, guarantees that after an arbitrary sequence of protocol steps all secrecy properties will hold, and if a receive event is then invoked, its associated authentication properties will also hold. For verification to be meaningful we need to make sure that all the security requirements of interest are formalized as either invariants of the log or postconditions of receive events, and the set of attacker capabilities is not too restrictive. In our approach those specifications are expressed explicitly and straightforwardly, which reduces the probability of a modeling error. Auxiliary invariants, on the other hand, can be added and corrected gradually: VCC will check if they are erroneous or insufficient.

A send event (Create in our example) is modelled by a ghost function that adds terms to the protocol set, publishes some of them in the attacker set and possibly updates the

```
void create_0(Term parent, Term obj)
  _(requires log.loaded[parent])
  _(requires \forall Term t;
    subterms(obj)[t] ==> !used(t))
  _(updates &log)
  _(ensures log.attacker[pub(obj)] &&
    log.attacker[Cipher(private_term(obj),
        symkey(parent))])
{
  _(unwrapping &log) {
    log.objects[obj] = \true;
    log.parent[obj] = parent;
    log.protocol = set_union(
        log.protocol, subterms(obj));

    Term enc_private = Cipher(
        private_term(obj),
        symkey(parent));
    log.protocol = set_union(
        log.protocol, subterms(enc_private));

    log.attacker[pub(obj)] = \true;
    log.attacker[enc_private] = \true;
  }
}
```

Listing 7: Create event on L0

```
void load_0(Term parent, Term obj)
  _(requires log.loaded[parent])
  _(requires log.attacker[pub(obj)] &&
    log.attacker[Cipher(private_term(obj),
        symkey(parent))])
  _(updates &log)
  // Authentication postcondition:
  _(ensures \old(log.objects)[obj] &&
    \old(log.parent)[obj] == parent)
{
  _(unwrapping &log)
    log.loaded[obj] = \true;
}

void att_decrypt_0(Term t, Term k)
  _(requires log.attacker[Cipher(t, k)])
  _(requires log.attacker[k])
  _(updates &log)
  _(ensures log.attacker[t])
{
  _(unwrapping &log)
    log.attacker[t] = \true;
}
```

Listing 8: Load event and attacker's decryption capability on L0

internal state (Listing 7). It might seem counterintuitive that create_0 receives the new object as an argument. In fact, the actual generation of a fresh storage key happens in the physical code; this key is then passed to create_0, whose only purpose is to register the key the log.

A receive event, such as Load, does not change the protocol and attacker sets, but might update the internal state. The event's precondition states that the message must be a part of the attacker knowledge, which models the fact that all communications in the system go through the attacker. In our example (Listing 8) the event is equipped with an authentication postcondition, which states that the loaded object belongs to the TPM hierarchy (and thus was sent by the Create event, as no other event can modify the hierarchy), and it is loaded under its initial parent.

A Dolev-Yao attacker is usually modelled as a set of deduction rules that transitively define the set of messages he can construct from some initial set. We encode those rules as events that add terms to the attacker set, with the premise of the rule corresponding to the event's precondition and the conclusion of the rule corresponding to the postcondition. For example, Listing 8 shows an event that models attacker's capability to decrypt a ciphertext once he knows the key.

For our case study we used the standard Dolev-Yao rules: generating a fresh literal, construction and destruction of compound terms, encryption and decryption, hashing. We also encoded several non-standard attacker capabilities in order to relax overly strong assumptions of symbolic cryptography. One of them is encrypting a fresh literal with a key not known to the attacker; it models the situation when the attacker provides an honest agent with a random string, when an encryption is expected. The other one is decomposing an encryption of a compound term with an unknown key into two encryptions and vice versa; this event models the distributivity of stream and block ciphers over concatenation.

Verifying correctness of the events requires adding auxiliary invariants to the log. While these invariants are checked by VCC and thus do not need to be trusted (they cannot accidentally weaken the security invariants or the attacker model), getting them right is a non-trivial task. Based on our experience, we can suggest the following invariant patterns:

1. *Dolev-Yao rules* describing how the attacker could have learnt a particular term. For instance an invariant **forall** Term t, k; attacker[Cipher(t, k)] ==> protocol[Cipher(t, k)] || (attacker[t] && attacker[k]) says that he can learn a ciphertext by either eavesdropping it or constructing it from a plaintext and a key he knows. Those invariants do not depend on the protocol, but only on the attacker model and the term algebra (and thus reusable).
2. Invariants stating that the protocol set and the used set are closed under addition of subterms (also protocol independent).
3. *Message format* invariants, describing the shape of the messages generated by honest agents. For example an invariant **forall** Term t, k; protocol[Cipher(t, k)] ==> is_object_sym_key(**this**, k) says that honest agents only produce encryptions with secret symmetric keys. These invariants are in general not reusable, but can be derived straightforwardly from the protocol. Note that there is no harm in making them stronger than required by adding all the knowledge about the protocol messages.
4. *Internal data* invariants, for example saying that a public key of an object in the hierarchy never coincides with a private key of the same or another object. These invariants are protocol-specific and most of the time have to be deduced from verification errors.
5. Additional *attacker restrictions* that do not correspond directly to the security goals. These are protocol-dependent and usually tricky to figure out. For example, we had to state that honest agents never publish a plaintext private area: **forall** Term t1, t2; protocol[Private(t1, t2)] ==> !attacker[Private(t1, t2)], because the protocol security relies on the fact that private areas are only sent encrypted.

4.3 From Term Algebra to Byte Strings (L1)

The initial model represents protocol messages as symbolic terms, while in the physical code messages are plain byte strings. In order to connect the two descriptions, we need a means to match a term to its string representation and vice versa. To this end, we have developed a VCC library containing a ghost type ByteString that encodes finite sequences of bytes of arbitrary size, together with a number of specification functions manipulating those sequences. For example, the function from_array(BYTE *data, **integer** size) returns a byte string stored in a physical byte array.

```
_(ghost typedef struct {
  // Set of used strings:
  bool strings[ByteString];
  // Mapping from strings to terms:
  Term term[ByteString];

  // Ownership:
  _(invariant \mine(&log))
  // Gluing invariants:
  _(invariant \forall ByteString s;
    strings[s] ==> used(log, term[s])
      && string(term[s]) == s)
  _(invariant \forall Term t;
    used(log, t) ==> strings[string(t)]
      && term[string(t)] == t)
} Table)
_(ghost Table table)
```

Listing 9: Representation table

```
void att_compute_hash_1(ByteString s)
  _(requires attacker_string(s))
  _(updates &table)
  _(ensures attacker_string(lib_hash(s)))
{
  // Symbolic assumption:
  _(assume table.strings[lib_hash(s)] ==>
    is_hash(table.term[lib_hash(s)]) &&
    string(hash_arg(
        table.term[lib_hash(s)])) == s)

  _(unwrapping &table) {
    att_compute_hash_0(table.term[s]);
    add(Hash(table.term[s]));
  }
}
```

Listing 10: Attacker's hashing capability on L1

Matching terms to strings is straightforward, as each term has a unique string representation. We introduce a specification function ByteString string(Term t) that defines the string format for every term constructor in such a way that it corresponds to the physical code. As for cryptographic terms (Cipher and Hash), we assume that the implementation uses a trusted cryptographic library to compute the corresponding byte strings, and its specification and verification is outside of the scope of our problem. Thus we model these operations with uninterpreted functions lib_encrypt and lib_hash that operate on values of type ByteString.

The other direction — mapping strings to terms — cannot be expressed as a function for cardinality reasons (e.g., hashing is generally not injective, and a ciphertext can in principle coincide with a hash value). To be able to apply the symbolic model of cryptography to byte string messages, following [8], we make *symbolic assumptions* on string-manipulating operations:

– an operation that corresponds to constructing a new term, cannot produce a string that had been used before as a representation of a different term;
– if an operation requires the corresponding term to be of a particular type, it cannot be performed on a string that represents a term of a different type (for example, a string that is obtained as a hash cannot be decrypted).

In the code instances of these assumptions appear in inline **assume** statements of L1 events (see examples below).

Symbolic assumptions guarantee that string has no collisions within the set of terms used in a protocol execution, and thus there exists a mapping from used string to used terms. This mapping, together with the set of used strings and the obvious gluing invariant is stored in a data structure called *representation table* (Listing 9).

The set of events of L1 closely corresponds to that of L0, except that they are specified using byte strings rather than terms. Following the general approach of Sect. 4.1,

```
_(dynamic_owns) typedef struct {
  OBJECT_SLOT slots[MAX_LOADED];

  _(invariant \forall \integer i;
    0 <= i && i < MAX_LOADED
      ==> \mine(&slots[i]))
  _(invariant \mine(&table))
  // Gluing invariant:
  _(invariant \forall \integer i;
    0 <= i && i < MAX_LOADED &&
    slots[i].occupied ==>
    log.loaded[term(&slots[i].object)])
} TPM_STORAGE;
TPM_STORAGE storage;
```

Listing 11: Physical state of the TPM

```
typedef struct {
  UINT16 keySize;
  BYTE key[MAX_SYM_DATA];
  _(ghost ByteString content)
  _(invariant
      keySize <= MAX_SYM_DATA)
  _(invariant
      content ==
      from_array(key, keySize))
} SYM_KEY;
```

Listing 12: Physical representation of a symmetric key

each refined event calls its L0 counterpart to modify the log and then updates the representation table accordingly. As an example let us consider the attacker's hashing capability (Listing 10). The attacker_string predicate states that a string represents a term known to the attacker. The add(Term) function adds a term and the corresponding string to the representation table, provided the no-collision condition holds: the string is not yet associated with another term. To satisfy this condition we have to add a symbolic assumption to the body of att_compute_hash_1. It states that if lib_hash(s) already occurs in the representation table, its corresponding term is a Hash, and moreover the argument of the hash can only map to s (i.e., we did not encounter a collision of lib_hash).

Symbolic assumptions are weaker for terms whose string representation is indeed injective. For example, when adding an encryption lib_encrypt(s, k) it is sufficient to assume that the corresponding term, if in the table, is a Cipher and its key maps to k; we can then prove that its plaintext also maps to s.

Sound handling of compound terms requires their string representation to be injective not only in both parts of the compound, but also in its type. Essentially, one has to verify for the message format used in the physical code that for any byte string there is at most one way to parse it into a compound term. With this property, the only symbolic assumption that is needed to add a compound term to the table is that its representation does not coincide with the representation of any literal, encryption or hash, which is reasonable. Without relying on injectivity it is hard to justify absence of collisions among compound terms.

In general, one has to be careful with symbolic assumptions, as they are a part of the trusted specification. Similarly to [8], our first refinement makes those assumptions *explicit*, which simplifies their informal validation.

4.4 Physical Code (L2)

The concrete variables of the second refinement are the global variables of the physical code. In our case, TPM stores a list of loaded objects (as an array of object slots that can be either occupied or empty). We add a gluing invariant connecting this array to the loaded set of the log (Listing 11).

```
TPM_RC Create(UINT32 parentHandle, PUBLIC *public, PRIVATE *private)
  _(requires object_exists(parentHandle))
  _(updates &storage)
  _(ensures attacker_string(public->content)) // "Gluing" postcondition
  _(ensures attacker_string(private->content)) // "Gluing" postcondition

TPM_RC Load(UINT32 parentHandle, PRIVATE *private, PUBLIC *public,
  UINT32 *objectHandle)
  _(requires object_exists(parentHandle))
  _(requires attacker_string(public->content)) // "Gluing" precondition
  _(requires attacker_string(private->content)) // "Gluing" precondition
  _(updates &storage)
  // Authentication postcondition:
  _(ensures \result == SUCCESS ==>
    \old(log.objects)[term(storage.slots[*objectHandle].object)] &&
    \old(log.parent)[term(storage.slots[*objectHandle].object)] ==
      term(storage.slots[parentHandle].object))

void SymmetricEncrypt(UINT16 keyBits, BYTE *key, UINT16 dataSize, BYTE* data)
    _(ensures from_array(data, dataSize) ==
        lib_encrypt(\old(from_array(data, dataSize)), from_array(key, keyBits / 8)))
```

Listing 13: Some contracts of Create and Load and an extract from the cryptographic library

A TPM object is represented in the code by an instance of the OBJECT struct, which contains instances of PUBLIC and SENSITIVE. As expected, PUBLIC stores the public asymmetric key, while SENSITIVE stores the secret asymmetric key and an instance of SYM_KEY, which in turn stores the symmetric protection key. All keys are stored in statically allocated buffers with a length field (see Listing 12 for an example). To access the byte string stored in a buffer we add a ghost field content to all structs that contain such buffers.

Note that there is no physical data structure representing the attacker knowledge, which could be connected by a gluing invariant to the attacker set of the log. Instead all the information flowing in and out of the TPM should be considered known to the attacker. This property can be encoded in the pre- and postconditions of TPM functions that communicate with the outside world.

For an example, let us look at Create (Listing 13). Its signature reveals that it can pass information to the outside world through two buffers: public and private. Thus it has to be equipped with a *gluing postcondition*, stating that the content of each output buffer corresponds to a term known to the attacker. An intuition behind this postcondition is that whatever Create returns to the outside world is safe to publish, because, according to the results of L0, the attacker cannot use it to violate the security goals.

The gluing postcondition forces Create to update the log and the representation table, which is done by invoking its L1 counterpart, create_1. To make the connection between the two levels, we have to verify that the protocol messages computed by the physical code are the same as described by the ghost code. To achieve that for cryptographic terms Cipher and Hash we have to instrument the cryptographic library with contracts in terms of the uninterpreted functions lib_encrypt and lib_hash introduced in L1 (see an example in Listing 13).

Our second TPM command, Load (Listing 13), receives data from the outside world; thus the content of its input buffers has to be related to the attacker knowledge through *gluing preconditions*. Note that these preconditions have somewhat special semantics: they do not express a contract with the client code, but rather our model of the client.

The gluing invariant of storage forces Load to update the loaded set of the log, which is accomplished through a call to load_1. The latter requires not only that the input strings be known to the attacker, but also that they have a correct format (in this case: that the integrity value matches). Those preconditions are established by the physical code that parses the input buffers and performs the integrity check.

We do not provide physical code for the attacker events on L2, because a concrete implementation of a TPM client would be of limited usefulness for our problem. Rather we would like to argue that whatever program a client can write using the given TPM interface, can be proved to refine a sequence of L1 events. This argument though remains informal and comes down to adequacy of the chosen attacker model.

Our technique has several benefits when it comes to the physical code level. First, it does not pose any requirements on the structure of the program and thus can work with pre-existing code. Second, the security-related specification only appears in a few top-level functions; the rest of the code is only concerned with memory safety and functional correctness, which considerably improves scalability.

5 Empirical Results

In this section we summarize the results of our case study, present some statistics and share our observations.

In the case study we managed to verify a slightly simplified version of the draft reference implementation of two TPM 2.0 commands, Create and Load, against the two security goals described in Sect. 3. Even though our objective was thorough verification of the protocol code, as opposed to finding bugs, we still discovered two faults in the original implementation, one related to memory safety and another one to security.

The security fault resided in the code performing the integrity check before loading a key. In the faulty version, if the first two bytes of the decrypted private area were zero, the integrity check was skipped and the rest of the private buffer was treated as the sensitive area. In this case, in order to load an incorrect object the attacker has to guess a value for the private area, such that its first two bytes would decrypt to zeros and the rest would fit the format of a sensitive area (which is easier than matching the integrity value). This fault shows up during verification as a failure to prove a postcondition of the function performing the integrity check, which in turn is needed to establish the precondition of load_1 and eventually to update the log in a consistent way. Note that in this case the error lies in the implementation rather than in the protocol itself, and thus could not be discovered by a high-level protocol checking tool.

One of the reasons we had to simplify the code is that VCC had problems reasoning about deep syntactic nesting of C structs, which TPM uses to store parameters and attributes of objects. Our simplified implementation only works with storage keys and supports just one encryption and one hashing algorithm, which eliminates the need to store most of the object parameters. As a consequence we also removed the code that checks consistency of those parameters or works explicitly with other types of objects.

File	Specs	Code	Ratio
ByteString.h	127		
CryptUtil.h	47	31	151%
TPM_Types.h	43	37	116%
marshal.h	94	32	293%
marshal.c	33	199	16%
create_load_0.c	384		
create_load_1.c	488		
create_load_2.c	282	205	137%
TOTAL	1498	504	297%

The table in this section gives code metrics for our case study. The annotation overhead for files containing both physical code and specification is usually around 150%, which is consistent with previous results for VCC. However, we also have two extra "implementations" of the protocol containing just ghost code, which brings the overall overhead to about 300%. The overhead in [8] is roughly 150%, but does not include the Coq proofs. Note, that our refinement models should not be understood as just overhead, as they convey useful information about the system in an easy to understand, operational way, where the hints for the verifier only comprise a fraction of the code.

Running VCC on the whole case study takes about 120 seconds (on a slightly dated 2.8GHz Core2 CPU using a single core), and only one function takes longer than 10 seconds to verify (precisely 38 seconds), whereas two more take between 3 and 10. They are all physical level functions, which involve multiple struct assignments, numerous function calls reading and writing entire nested structs, and complex postconditions. All other functions (including all the ghost-only functions) take less than 3 seconds to verify. It thus seems that handling of relatively complex C structs in VCC needs to be improved, whereas reasoning about pure mathematical data structures (even if they involve complex invariants) works well.

In terms of development time, specification and verification effort can be estimated as 4 person-months, including studying the TPM source code, developing the refinement approach and writing reusable specifications (e.g., the byte string library).

The major verification road-blocker, and source of frustration, was understanding failed verification attempts when working with physical code, especially large functions that mutate a lot of state. One reason is the lack of immediate feedback: when verifying those functions VCC would rarely produce a quick negative result, but rather keep on running for indefinite time. The Z3 Inspector [5] tool, monitoring progress of the back-end proof engine, was invaluable in those cases. Another reason is that error reports are often related to internal properties of the VCC memory model and are obscure to a non-expert user, as compared with errors expressed in terms of user-defined specifications.

6 Related Work

The introduction (Sect. 1) compares our work with previous work [8] on using VCC for security verification, which was in turn based on *invariants for cryptographic structures* [3].

There exist special-purpose tools for verifying security properties of C code, using abstract interpretation, like Csur [9], or model-checking techniques, as in ASPIER [4].

ASPIER only considers a limited number of protocol sessions, whereas Csur does not prove memory safety.

A number of tools use various static analysis and symbolic execution techniques to extract protocol description from code, or check conformance of code with a specific protocol. These tools are useful for finding bugs but their usability for sound verification is limited. In particular, various static analysis techniques tend to fail when confronted with slightly unusual or more complex codebases. On the other hand, in VCC proving correctness of complex code is a matter of putting in enough annotations, which is admittedly a difficult but manageable task.

Stepwise refinement has been used before to systematically construct cryptographic protocols from security goals [14]. Our approach complements this work, starting from a symbolic protocol definition (the final result of their technique) and refining it even further into an implementation.

There are other examples of encoding refinement techniques within a general-purpose program verifier [11], however they have mostly been applied to constructing relatively simple algorithms, rather than verifying pre-existing real-world software systems.

We believe that our approach could be implemented in any verification environment with expressive enough specification language. For the C programming language this includes Frama-C [12], VeriFast [10], and KeyC [13].

7 Conclusions

We proposed a novel approach to verifying implementations of security protocols based on stepwise refinement. To this end we developed an encoding of refinement in a general-purpose program verifier, which we believe can also be used in other problem domains.

Our refinement strategy for security protocols separates specification and verification into three levels of abstraction. Security goals can be expressed straightforwardly and concisely on the most abstract level. In general, all the specifications that have to be trusted (validated against the informal requirements) are explicit and relatively concise. They include security invariants, pre- and postconditions of events, gluing invariants between different levels, and symbolic assumptions. All other annotations are checked by the verifier, which makes our approach less error-prone.

The proposed technique is flexible and scalable enough to handle real pre-existing C code, which we confirmed by applying it to the draft reference implementation of TPM 2.0.

One direction of future work is extending the TPM case study to remove the code simplifications and include more TPM commands. The auxiliary invariants of the log would need to be extended to allow the additional command behavior, and the existing commands would need to be reverified against those invariants.

Another direction is applying the refinement approach to other security protocols. In a network-based protocol there is no shared physical state between the participants, however the ghost state can still be shared, which enables the use of our approach. In multi-message protocols honest agents and the attacker have to be executed concurrently. This does not affect the ghost code, as ghost events represent atomic actions of sending and receiving single messages. A physical function that implements a protocol role, can call multiple ghost events and havoc the ghost state in between. Because of the flexibility of the general-purpose verifier, we believe that our approach can be naturally extended to handle other kinds of security properties and attacker models.

Acknowledgements. We appreciate the collaboration of François Dupressoir, Paul England, Cédric Fournet, Andy Gordon and David Wooten on the TPM project. We are grateful to François Dupressoir, Carlo Furia and Andy Gordon, as well as the anonymous referees, for their comments on the draft versions of this paper.

References

1. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundam. Inform. 77(1-2), 1–28 (2007)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: POPL 2010, pp. 445–456. ACM, New York (2010)
4. Chaki, S., Datta, A.: Aspier: An automated framework for verifying security protocol implementations. In: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, pp. 172–185. IEEE Computer Society, Washington, DC, USA (2009)
5. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
8. Dupressoir, F., Gordon, A.D., Jürjens, J., Naumann, D.A.: Guiding a general-purpose C verifier to prove cryptographic protocols. In: IEEE Computer Security Foundations Symposium (2011)
9. Goubault-Larrecq, J., Parrennes, F.: Cryptographic Protocol Analysis on Real C Code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005)
10. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
11. Leino, K.R.M., Yessenov, K.: Automated stepwise refinement of heap-manipulating code (2010)
12. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. PhD thesis, Université Paris-Sud (January 2009)
13. Mürk, O., Larsson, D., Hähnle, R.: KeY-C: A Tool for Verification of C Programs. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 385–390. Springer, Heidelberg (2007)
14. Sprenger, C., Basin, D.A.: Developing security protocols by refinement. In: ACM Conference on Computer and Communications Security, pp. 361–374 (2010)

Deciding Functional Lists with Sublist Sets

Thomas Wies[1], Marco Muñiz[2], and Viktor Kuncak[3]

[1] New York University, New York, NY, USA
[2] University of Freiburg, Germany
[3] EPFL, Switzerland

Abstract. Motivated by the problem of deciding verification conditions for the verification of functional programs, we present new decision procedures for automated reasoning about functional lists. We first show how to decide in NP the satisfiability problem for logical constraints containing equality, constructor, selectors, as well as the transitive sublist relation. We then extend this class of constraints with operators to compute the set of all sublists, and the set of objects stored in a list. Finally, we support constraints on sizes of sets, which gives us the ability to compute list length as well as the number of distinct list elements. We show that the extended theory is reducible to the theory of sets with linear cardinality constraints, and therefore still in NP. This reduction enables us to combine our theory with other decidable theories that impose constraints on sets of objects, which further increases the potential of our decidability result in verification of functional and imperative software.

1 Introduction

Specifications using high-level data types, such as sets and algebraic data types have proved effective for describing the behavior of functional and imperative programs [12,26]. Functional lists are particularly convenient and widespread in both programs and specifications. Efficient decision procedures for reasoning about lists can therefore greatly help automate software verification tasks.

Theories that allow only constructing and decomposing lists correspond to term algebras and have efficient decision procedures for quantifier-free fragments [1,15]. However, these theories do not support list concatenation or sublists. Adding list concatenation makes the logic difficult because it subsumes the existential problem for word equations [11,17,7], which has been well-studied and is known to be difficult.

This motivates us to use as a starting point the logic of lists with a sublist (suffix) relation, which can express some (even if not all) of the properties expressible using list concatenation. We give an axiomatization of this theory where quantifiers can be instantiated in a complete and efficient way, following the methodology of local theory extensions [18]. Although local theory extensions have been applied to term algebras with certain recursive functions [19], they have not been applied to term algebras in the presence of the sublist operation.

The general subterm relation in term algebras was shown to be in NP using different techniques [21], without discussion of practical implementation procedures and without support for set operators. Several expressive logics of linked imperative data structures have been proposed [3,9,23,10]. In these logics, variables range over graph nodes, as opposed to lists viewed as terms. In other words, the theories that we consider have an additional extensionality axiom, which ensures that no two list objects in the universe have identical tail and head. This axiom has non-trivial consequences on the set of satisfiable formulas and requires a new decision procedure. Our logic admits reasoning about set-algebraic constraints, as well as cardinalities of sublist and content sets. Note that the cardinality of the set of sublists of a list xs can be used to express the length xs. Our result is particularly interesting given that the theory of concatenation with word length function is not known to be decidable [4]. Decidable logics that allow reasoning about length of lists have been considered before [20]. However, our set-algebraic constraints are strictly more expressive and capture forms of quantification that are useful for the specification of complex properties.

Contributions. We summarize the contributions of our paper as follows:

- We give a set of local axioms for the theory of lists with sublist relation that admits an efficient implementation in the spirit of [9,23] and can leverage general implementation methods for local theory extensions [5,6].
- We show how to extend this theory with an operator to compute the longest common suffix of two lists. We also give local axioms that give the decision procedure for the extended logic.
- We show how to further extend the theory by defining sets of elements that correspond to all sublists of a list, and then stating set algebra and size operations on such sets. Using a characterization of the models of this theory, we establish that the theory admits a reduction to the logic BAPA of sets with cardinality constraints [8]. We obtain a decidable logic that supports reasoning about the contents of lists as well as about the number of elements in a list.

Impact on Verification Tools. We have found common functions in libraries of functional programming languages and interactive theorem provers that can be verified to meet a detailed specification using our logic. We discuss several examples in the paper. Moreover, the reduction to BAPA makes it possible to combine this logic with a number of other BAPA-reducible logics [24,20,25,16]. Therefore, we believe that our logic will be a useful component of verification tools in the near future.

An extended version of this paper with proofs and additional material is available as a technical report [22].

2 Examples

We describe our contributions through several examples. In the first two examples we show how we use our decision procedure to verify functional correctness

```
def drop[T](n: Int, xs: List[T]): List[T] = {
  if (n ≤ 0) xs
  else xs match {
    case nil ⇒ nil
    case cons(x, ys) ⇒ drop(n−1, ys)
  }
} ensuring (zs ⇒ (n < 0 → zs = xs) ∧ (n ≥ 0 ∧ length(xs) < n → zs = nil) ∧
                 (n ≥ 0 ∧ length(xs) ≥ n → zs ⪯ xs ∧ length(zs) = length(xs) − n))
```

Fig. 1. Function drop that drops the first n elements of a list xs

$$\begin{aligned}&\mathsf{n} > 0 \;\wedge\; \mathsf{xs} \neq \mathsf{nil} \;\wedge\; \mathsf{cons}(\mathsf{x},\mathsf{ys}) = \mathsf{xs} \;\wedge\; \mathsf{zs} \preceq \mathsf{ys} \;\wedge\\ &(\mathsf{n}-1 \geq 0 \wedge \mathsf{length}(\mathsf{ys}) \geq \mathsf{n}-1 \;\rightarrow \mathsf{length}(\mathsf{zs}) = \mathsf{length}(\mathsf{ys}) - (\mathsf{n}-1)) \rightarrow\\ &\quad \mathsf{n} \geq 0 \wedge \mathsf{length}(\mathsf{xs}) \geq \mathsf{n} \rightarrow \mathsf{length}(\mathsf{zs}) = \mathsf{length}(\mathsf{xs}) - \mathsf{n}\end{aligned}$$

Fig. 2. One of the verification conditions for the function drop

of a function written in a functional programming notation similar to the Scala programming language [14]. In our third example we demonstrate the usefulness of our logic to increase the degree of automation in interactive theorem proving. Throughout this section we use the term sublist for a suffix of a list.

Example 1: Dropping Elements from a List. Our first example, listed in Figure 1, is the function drop of the List class in the Scala standard library (such functions also occur in standard libraries for other functional languages, such as Haskell). The function takes as input an integer number n and a parametrized functional list xs. The function returns a functional list zs which is the sublist obtained from xs after dropping the initial n elements.

The **ensuring** statement specifies the postcondition of the function (a precondition is not required). The postcondition is expressed in our logic FLS^2 of functional lists with sublist sets shown in Figure 8. We consider the third conjunct of the postcondition in detail: it states that if the input n is a positive number and smaller than the length of xs then (1) the returned list zs is a sublist of the input list xs, denoted by $\mathsf{zs} \preceq \mathsf{xs}$, and (2) the length of zs is equal to the length of xs discounting the n dropped elements.

Deciding Verification Conditions. To verify the correctness of the drop function, we generate verification conditions and use our decision procedure to decide their validity. Figure 2 shows one of the generated verification conditions, expressed in our logic of Functional Lists with Sublist Sets (FLS^2). This verification condition considers the path through the second case of the **match** expression.

The verification condition can be proved valid using the FLS^2 decision procedure presented in Section 7. The theory FLS^2 is a combination of the theory FLS and the theory of sets with linear cardinality constraints (BAPA). Our decision procedure follows the methodology of [24] that enables the combination of such set-sharing theories via reduction to BAPA. Figure 3 illustrates how this decision procedure proves subgoal G_2. We first negate the subgoal and then eliminate the length function. For every list xs we encode its length length(xs) using sublist sets as follows. We introduce a set variable Xs and define it as the set of all sublists of

FLS fragment:
$Xs = \sigma(\mathsf{xs}) \land Ys = \sigma(\mathsf{ys}) \land Zs = \sigma(\mathsf{zs}) \land$
$\mathsf{xs} \neq \mathsf{nil} \land \mathsf{cons}(\mathsf{x}, \mathsf{ys}) = \mathsf{xs} \land \mathsf{zs} \preceq \mathsf{ys}$
Projection onto shared sets Xs, Ys, Zs:
$Zs \subseteq Ys \ \land \ Ys \subseteq Xs \ \land \ \mathsf{card}(Xs) > 1 \ \land \ \mathsf{card}(Xs) = \mathsf{card}(Ys) + 1$

BAPA fragment:
$xs\_length = \mathsf{card}(Xs) - 1 \ \land \ ys\_length = \mathsf{card}(Ys) - 1 \ \land \ zs\_length = \mathsf{card}(Zs) - 1 \land$
$\mathsf{n} > 0 \ \land \ (\mathsf{n} - 1 \geq 0 \ \land \ ys\_length \geq \mathsf{n} - 1 \rightarrow zs\_length = ys\_length - (\mathsf{n} - 1)) \land$
$\mathsf{n} \geq 0 \ \land \ xs\_length \geq \mathsf{n} \ \land \ zs\_length \neq xs\_length - \mathsf{n}$
Projection onto shared sets Xs, Ys, Zs: $\mathsf{card}(Xs) \neq \mathsf{card}(Ys) + 1$

Fig. 3. Separated conjuncts for the negated subgoal G_2 of the VC in Figure 2 with the projections onto shared sets

xs: $\{l.\, l \preceq \mathsf{xs}\}$, which we denote by $\sigma(\mathsf{xs})$. We then introduce an integer variable xs_length that denotes the length of xs by defining $xs\_length = \mathsf{card}(Xs) - 1$, where $\mathsf{card}(Xs)$ denotes the cardinality of set Xs. Note that we have to subtract 1, since nil is also a sublist of xs. We then purify the resulting formula and separate it into two conjuncts for the individual fragments. These two conjuncts are shown in Figure 3. The two separated conjuncts share the set variables Xs, Ys, and Zs. After the separation the underlying decision procedure of each fragment computes a projection of the corresponding conjunct onto the shared set variables. These projections are the strongest BAPA consequences that are expressible over the shared sets in the individual fragments. After the projections have been computed, we check satisfiability of their conjunction using the BAPA decision procedure. In our example the conjunction of the two projections is unsatisfiable, which proves that G_2 is valid. In Section 7 we describe how to construct these projections onto set variables for the FLS^2 theory.

Example 2: Greatest Common Suffix. Figure 4 shows our second example, a Scala function gcs, which takes as input two functional lists xs, ys and their corresponding lengths lxs, lys. This precondition is specified by the **require** statement. The function returns a pair (zs,lzs) such that zs is the greatest common suffix of the two input lists and lzs its length. This is captured by the postcondition. Our logic provides the operator $\mathsf{xs} \sqcap \mathsf{ys}$ that denotes the greatest common suffix of two lists xs and ys. Thus, we can directly express the desired property.

Figure 5 shows one of the verification conditions that are generated for the function gcs. This verification condition captures the case when the lists xs, ys are not empty, their lengths are equal, their head elements x, y are equal, and lz1s is equal to length(xs)-1. The verification condition can again be split into two subgoals. We focus on subgoal G_1. Figure 6 shows the separated conjuncts for this subgoal and their projections onto the shared set variables Xs, Ys, Zs, and $Z1s$. Using the BAPA decision procedure, we can again prove that the conjunction of the two projections is unsatisfiable.

Example 3: Interactive Theorem Proving. Given a complete specification of functions such as drop and gcd in our logic, we can use our decision procedure to automatically prove more complex properties about such functions. For instance,

```
def gcs[T](xs: List[T], lxs: Int, ys: List[T], lys: Int): (List[T], Int)
require (length(xs)=lxs ∧ length(ys)=lys) =
  (xs,ys) match {
    case (nil, _) ⇒ (nil, 0)
    case (_, nil) ⇒ (nil, 0)
    case (cons(x, x1s), cons(y, y1s)) ⇒
      if (lxs > lys) gcs(x1s, lxs−1, ys, lys)
      else if (lxs < lys) gcs(xs, lxs, y1s, lys−1)
      else {
        val (z1s, lz1s) = gcs(x1s, lxs−1, y1s, lys−1)
        if (x = y ∧ lz1s = (lxs − 1)) (cons(x, z1s), lz1s+1) else (z1s, lz1s)
      }
  } ensuring ((zs, lzs) ⇒ length(zs) = lzs ∧ zs = xs ⊓ ys)
```

Fig. 4. Function gcs that computes the greatest common suffix of two lists

$$\begin{aligned}&\mathsf{length}(\mathsf{xs}) = \mathsf{lxs} \;\wedge\; \mathsf{length}(\mathsf{ys}) = \mathsf{lys} \;\wedge\; \mathsf{xs} \neq \mathsf{nil} \;\wedge\; \mathsf{ys} \neq \mathsf{nil} \;\wedge\; \mathsf{lxs} = \mathsf{lys} \wedge \mathsf{x} = \mathsf{y} \wedge\\ &\mathsf{cons}(\mathsf{x},\mathsf{x1s}) = \mathsf{xs} \;\wedge\; \mathsf{cons}(\mathsf{y},\mathsf{y1s}) = \mathsf{ys} \;\wedge\; \mathsf{lz1s} = \mathsf{lxs} - 1 \wedge\\ &\mathsf{length}(\mathsf{z1s}) = \mathsf{lz1s} \;\wedge\; \mathsf{z1s} = \mathsf{xs1} \sqcap \mathsf{y1s} \;\wedge\; \mathsf{zs} = \mathsf{cons}(\mathsf{x},\mathsf{z1s}) \;\wedge\; \mathsf{lzs} = \mathsf{lz1s} + 1 \;\rightarrow\\ &\qquad \underbrace{\mathsf{length}(\mathsf{zs}) = \mathsf{lzs}}_{G_1} \;\wedge\; \underbrace{\mathsf{zs} = \mathsf{xs} \sqcap \mathsf{ys}}_{G_2}\end{aligned}$$

Fig. 5. One of the verification conditions for the function gcs

FLS fragment:
$Xs = \sigma(\mathsf{xs}) \wedge Ys = \sigma(\mathsf{ys}) \wedge Zs = \sigma(\mathsf{zs}) \wedge Z1s = \sigma(\mathsf{z1s}) \wedge$
$\mathsf{cons}(\mathsf{x},\mathsf{x1s}) = \mathsf{xs} \;\wedge \mathsf{cons}(\mathsf{y},\mathsf{y1s}) = \mathsf{ys} \wedge \mathsf{xs} \neq \mathsf{nil} \wedge \mathsf{x} = \mathsf{y} \wedge \mathsf{ys} \neq \mathsf{nil} \;\wedge$
$\mathsf{z1s} = \mathsf{x1s} \sqcap \mathsf{y1s} \wedge \mathsf{zs} = \mathsf{cons}(\mathsf{x},\mathsf{z1s})$
Projection onto shared sets Xs, Ys, Zs, $Z1s$:
$\mathsf{card}(Xs) > 1 \;\wedge\; \mathsf{card}(Ys) > 1 \;\wedge\; \mathsf{card}(Zs) > 1 \wedge$
$Z1s \subseteq Zs \;\wedge\; \mathsf{card}(Zs) = \mathsf{card}(Z1s) + 1 \wedge$
$((\mathsf{card}(Z1s) = \mathsf{card}(Xs) - 1 \vee \mathsf{card}(Z1s) = \mathsf{card}(Ys) - 1) \;\rightarrow\; Zs = Xs = Ys)$

BAPA fragment:
$xs\_length = \mathsf{card}(Xs) - 1 \;\wedge\; ys\_length = \mathsf{card}(Ys) - 1 \wedge$
$zs\_length = \mathsf{card}(Zs) - 1 \;\wedge\; z1s\_length = \mathsf{card}(Z1s) - 1 \wedge$
$xs\_length = \mathsf{lxs} \;\wedge\; ys\_length = \mathsf{lys} \;\wedge\; z1s\_length = \mathsf{lz1s} \wedge$
$\mathsf{lxs} = \mathsf{lys} \;\wedge\; \mathsf{lz1s} = \mathsf{lxs} - 1 \;\wedge\; \mathsf{lzs} = \mathsf{lz1s} + 1 \;\wedge\; zs\_length \neq \mathsf{lzs}$
Projection onto shared sets Xs, Ys, Zs, $Z1s$:
$\mathsf{card}(Z1s) = \mathsf{card}(Xs) - 1 \;\wedge\; \mathsf{card}(Z1s) = \mathsf{card}(Ys) - 1 \;\wedge\; \mathsf{card}(Zs) \neq \mathsf{card}(Z1s) + 1$

Fig. 6. Separated conjuncts for the negated subgoal G_1 of the VC in Figure 5 with projections onto the shared sets

the function drop is not just defined in the Scala standard library, but also in the theory List of the Isabelle/HOL interactive theorem prover [13]. Consider the following property of function drop:

$$n \leq m \rightarrow \tau(\mathsf{drop}(n,\mathsf{xs})) \subseteq \tau(\mathsf{drop}(m,\mathsf{xs}))$$

$$(n < 0 \to \mathsf{zs}_n = \mathsf{xs}) \land (n \geq 0 \land \mathsf{length}(\mathsf{xs}) < n \to \mathsf{zs}_n = \mathsf{nil}) \land$$
$$(n \geq 0 \land \mathsf{length}(\mathsf{xs}) \geq n \to \mathsf{zs}_n \preceq \mathsf{xs} \land \mathsf{length}(\mathsf{zs}_n) = \mathsf{length}(\mathsf{xs}) - n) \land$$
$$(m < 0 \to \mathsf{zs}_m = \mathsf{xs}) \land (m \geq 0 \land \mathsf{length}(\mathsf{xs}) < m \to \mathsf{zs}_m = \mathsf{nil}) \land$$
$$(m \geq 0 \land \mathsf{length}(\mathsf{xs}) \geq m \to \mathsf{zs}_m \preceq \mathsf{xs} \land \mathsf{length}(\mathsf{zs}_m) = \mathsf{length}(\mathsf{xs}) - m) \land$$
$$n \leq m \to \tau(\mathsf{zs}_n) \subseteq \tau(\mathsf{zs}_m)$$

Fig. 7. Lemma set_drop_subset_set_drop expressed in our logic

where the expression $\tau(\mathsf{xs})$ denotes the *content set* of a list xs, i.e., $\tau(\mathsf{xs}) = \{\mathsf{head}(l).\, l \preceq \mathsf{xs}\}$. This property corresponds to Lemma set_drop_subset_set_drop stated and proved in the Isabelle theory List. Using the postcondition of function drop to eliminate all occurrences of this function in Lemma set_drop_subset_set_drop yields the formula shown in Figure 7. This formula belongs to our logic. The proof of lemma set_drop_subset_set_drop that is presented inside the Isabelle theory is not fully automated, and involves the statement of an intermediate auxiliary lemma. Using our decision procedure, the main lemma can be proved directly and fully automatically, without requiring an auxiliary lemma. Our logic is, thus, useful to increase the degree of automation in interactive theorem proving.

3 Logic FLS^2 of Functional Lists with Sublists Sets

The grammar of our logic of functional lists with sublist sets is shown in Figure 8. It supports reasoning about lists built from list constructors and selectors, sublists, the length of lists, and cardinality and set algebraic constraints over the sets of sublists of lists $\sigma(l)$ as well their content sets $\tau(l)$.

The remainder of the paper is structured as follows. In Section 5 we first present the fragment FLS of the logic FLS^2 that allows formulas over lists and sublists, but not sets, cardinalities, or length constraints. We then formally define the semantics of the logic FLS and give a decision procedure for its satisfiability problem in Section 6. Finally, in Section 7 we show how to use this decision procedure for a BAPA reduction that decides the full logic FLS^2.

$$\begin{aligned}
F &::= A_L \mid A_S \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \\
A_L &::= T_L \preceq T_L \mid T_L = T_L \mid T_H = T_H \\
T_L &::= v_L \mid \mathsf{nil} \mid \mathsf{cons}(T_H, T_L) \mid \mathsf{tail}(T_L) \mid T_L \sqcap T_L \\
T_H &::= v_H \mid \mathsf{head}(T_L) \\
A_S &::= B_L = B_L \mid B_L \subseteq B_L \mid T_I = T_I \mid T_I < T_I \\
B_L &::= s_L \mid \emptyset \mid \{T_L\} \mid \sigma(T_L) \mid B_L \cup B_L \mid B_L \setminus B_L \\
B_H &::= s_H \mid \emptyset \mid \{T_H\} \mid \tau(T_L) \mid \mathsf{head}[B_L] \mid B_H \cup B_H \mid B_H \setminus B_H \\
T_I &::= v_I \mid K \mid T_I + T_I \mid K \cdot T_I \mid \mathsf{card}(B_L) \mid \mathsf{card}(B_H) \mid \mathsf{length}(T_L) \\
K &::= \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{aligned}$$

Fig. 8. Logic FLS^2 of lists, sublist, sublist sets, list contents, and size constraints

4 Preliminaries

In the following, we define the syntax and semantics of formulas. We also review the notions of partial structures and local theory extensions from [18].

Sorted Logic. We present our problem in sorted logic with equality. A *signature* Σ is a tuple (S, Ω), where S is a countable set of sorts and Ω is a countable set of function symbols f with associated arity $n \geq 0$ and associated sort $s_1 \times \cdots \times s_n \to s_0$ with $s_i \in S$ for all $i \leq n$. Function symbols of arity 0 are called *constant symbols*. We assume that all signatures contain the sort bool and for every other sort $s \in S$ a dedicated equality symbol $=_s \in \Omega$ of sort $s \times s \to \mathsf{bool}$. Note that we generally treat predicate symbols of sort $s_1, \ldots, s_n$ as function symbols of sort $s_1 \times \ldots \times s_n \to \mathsf{bool}$. Terms are built as usual from the function symbols in Ω and (sorted) variables taken from a countably infinite set X that is disjoint from Ω. A term t is said to be *ground*, if no variable appears in t. We denote by $\mathsf{Terms}(\Sigma)$ the set of all ground Σ-terms.

A Σ-*atom* A is a Σ-term of sort bool. We use infix notation for atoms built from the equality symbol. A Σ-*formula* F is defined via structural recursion as either one of A, $\neg F_1$, $F_1 \wedge F_2$, or $\forall x : s.F_1$, where A is a Σ-atom, F_1 and F_2 are Σ-formulas, and $x \in X$ is a variable of sort $s \in S$. We typically drop the sort annotation (both for quantified variables and the equality symbols) if this does not cause any ambiguity. We use syntactic sugar for Boolean constants (true, false), disjunctions ($F_1 \vee F_2$), implications ($F_1 \to F_2$), and existential quantification ($\exists x.F_1$). We define literals and clauses as usual. A clause C is called *flat* if no term that occurs in C below a predicate symbol or the symbol $=$ contains nested function symbols. A clause C is called *linear* if (i) whenever a variable occurs in two non-variable terms in C that do not start with a predicate or the equality symbol, the two terms are identical, and if (ii) no such term contains two occurrences of the same variable.

Total and Partial Structures. Given a signature $\Sigma = (S, \Omega)$, a *partial Σ-structure* α is a function that maps each sort $s \in S$ to a non-empty set $\alpha(s)$ and each function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \to s_0$ to a partial function $\alpha(f) : \alpha(s_1) \times \cdots \times \alpha(s_n) \rightharpoonup \alpha(s_0)$. If α is understood, we write just t instead of $\alpha(t)$ whenever this is not ambiguous. We assume that all partial structures interpret the sort bool by the two-element set of Booleans $\{0, 1\}$. We further assume that all structures α interpret the symbol $=_s$ by the equality relation on $\alpha(s)$. A partial structure α is called *total structure* or simply *structure* if it interprets all function symbols by total functions. For a Σ-structure α where Σ extends a signature Σ_0 with additional sorts and function symbols, we write $\alpha|_{\Sigma_0}$ for the Σ_0-structure obtained by restricting α to Σ_0.

Given a total structure α and a *variable assignment* $\beta : X \to \alpha(S)$, the evaluation $[\![t]\!]_{\alpha,\beta}$ of a term t in α, β is defined as usual. For a ground term t we typically write just $[\![t]\!]_\alpha$. A quantified variable of sort s ranges over all elements of $\alpha(s)$. From the interpretation of terms the notions of satisfiability, validity, and entailment of atoms, formulas, clauses, and sets of clauses in total structures are derived as usual. In particular, we use the standard interpretations for propositional connectives of classical logic. We write $\alpha, \beta \models F$ if α satisfies F under β where F is a formula, a clause, or a set of clauses. Similarly, we write $\alpha \models F$ if F is valid in α. In this case we also call α a *model* of F. The interpretation $[\![t]\!]_{\alpha,\beta}$ of a term t in a partial structure α is as for total structures, except that if $t = f(t_1, \ldots, t_n)$ for $f \in \Omega$ then $[\![t]\!]_{\alpha,\beta}$ is undefined if either $[\![t_i]\!]_{\alpha,\beta}$ is undefined for some i, or $([\![t_1]\!]_{\alpha,\beta}, \ldots, [\![t_n]\!]_{\alpha,\beta})$ is not in the domain of $\alpha(f)$. We say that a partial structure α *weakly satisfies* a literal L under β, written $\alpha, \beta \models_w L$, if (i) L is an atom A and either $[\![A]\!]_{\alpha,\beta} = 1$ or $[\![A]\!]_{\alpha,\beta}$ is undefined, or (ii) L is a negated atom $\neg A$ and either $[\![A]\!]_{\alpha,\beta} = 0$ or $[\![A]\!]_{\alpha,\beta}$ is undefined. The notion of weak satisfiability is extended to clauses and sets of clauses as for total structures. A clause C (respectively, a set of clauses) is *weakly valid* in a partial structure α if α weakly satisfies α for all variable assignments β. We then call α a *weak partial model* of C.

Theories and Local Theory Extensions. A *theory* $\mathcal{T}$ for a signature Σ is simply a set of Σ-formulas. We consider theories $\mathcal{T}(\mathcal{M})$ defined as a set of Σ-formulas that are valid in a given set of models $\mathcal{M}$, as well as theories $\mathcal{T}(\mathcal{K})$ defined as a set of Σ-formulas that are consequences of a given set of formulas $\mathcal{K}$. In the latter case, we call $\mathcal{K}$ the *axioms* of the theory $\mathcal{T}(\mathcal{K})$ and we often identify $\mathcal{K}$ and $\mathcal{T}(\mathcal{K})$.

In what follows, we consider theories that are defined by a set of axioms. Let $\Sigma_0 = (S, \Omega_0)$ be a signature and assume that signature $\Sigma_1 = (S, \Omega_0 \cup \Omega_1)$ extends Σ_0 by new function symbols Ω_1. We call the function symbols in Ω_1 *extension symbols* and terms starting with extension symbols *extension terms*. Now, a theory $\mathcal{T}_1$ over Σ_1 is an *extension* of a theory $\mathcal{T}_0$ over Σ_0, if $\mathcal{T}_1$ is obtained from $\mathcal{T}_0$ by adding a set of (universally quantified) clauses $\mathcal{K}$. In the following, when we refer to a set of ground clauses G, we assume they are over the signature $\Sigma_1^c = (S, \Omega_0 \cup \Omega_1 \cup \Omega_c)$ where Ω_c is a set of new constant symbols. Let $\mathcal{K}$ be a set of (universally quantified) clauses. We denote by $\mathsf{st}(\mathcal{K}, G)$ the set of all ground subterms that appear in $\mathcal{K}$ or G and by $\mathcal{K}[G]$ the set of all instantiations of clauses in $\mathcal{K}$ where variables appearing below extension terms have been instantiated by the terms in $\mathsf{st}(\mathcal{K}, G)$. Then an extension $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is a *local extension* if it satisfies condition (Loc):

(Loc) For every finite set of ground clauses G, $G \cup \mathcal{T}_1 \models \mathsf{false}$ iff there is no partial Σ_1^c-structure α such that $\alpha_{|\Sigma_0}$ is a total model of $\mathcal{T}_0$, all terms in $\mathsf{st}(\mathcal{K}, G)$ are defined in α, and α weakly satisfies $\mathcal{K}[G] \cup G$.

5 Logic FLS of Functional Lists with Sublists

We now define the logic of functional lists with sublists (FLS) and its accompanying theory. The logic FLS is given by all quantifier-free formulas over the signature $\Sigma_{\mathsf{FLS}} = (S_{\mathsf{FLS}}, \Omega_{\mathsf{FLS}})$. The signature Σ_{FLS} consists of sorts $S_{\mathsf{FLS}} = \{\mathsf{bool}, \mathsf{list}, \mathsf{data}\}$ and function symbols $\Omega_{\mathsf{FLS}} = \{\mathsf{nil}, \mathsf{cons}, \mathsf{head}, \mathsf{tail}, \sqcap, \preceq\}$. The sorts of the function symbols in Ω_{FLS} are shown in Figure 9. We use infix notation for the symbols $\sqcap$ and $\preceq$.

$\mathsf{nil} : \mathsf{list}$	$\mathsf{tail} : \mathsf{list} \to \mathsf{list}$	$\preceq : \mathsf{list} \times \mathsf{list} \to \mathsf{bool}$
$\mathsf{cons} : \mathsf{data} \times \mathsf{list} \to \mathsf{list}$	$\mathsf{head} : \mathsf{list} \to \mathsf{data}$	$\sqcap : \mathsf{list} \times \mathsf{list} \to \mathsf{list}$

Fig. 9. Sorts of function symbols in the signature Σ_{FLS}

$$\begin{aligned}
\alpha_{\mathsf{FLS}}(\mathsf{list}) &\overset{\text{def}}{=} \mathsf{L} \overset{\text{def}}{=} \{\, t \in \mathsf{Terms}(\Sigma_{\mathsf{L}}) \mid t : \mathsf{list} \,\} \\
\alpha_{\mathsf{FLS}}(\mathsf{data}) &\overset{\text{def}}{=} \mathsf{D} \overset{\text{def}}{=} \{\, t \in \mathsf{Terms}(\Sigma_{\mathsf{L}}) \mid t : \mathsf{data} \,\} \\
\alpha_{\mathsf{FLS}}(\mathsf{cons}) &\overset{\text{def}}{=} \mathsf{cons}_{\mathsf{L}} \overset{\text{def}}{=} \lambda(d,l).\, \mathsf{cons}(d,l) \\
\alpha_{\mathsf{FLS}}(\mathsf{nil}) &\overset{\text{def}}{=} \mathsf{nil} \\
\alpha_{\mathsf{FLS}}(\mathsf{tail}) &\overset{\text{def}}{=} \mathsf{tail}_{\mathsf{L}} \overset{\text{def}}{=} \lambda l.\, \text{if } l = \mathsf{nil} \ \text{ then } \mathsf{nil} \ \text{ else } l' \text{ where } l = \mathsf{cons}(d, l') \\
\alpha_{\mathsf{FLS}}(\mathsf{head}) &\overset{\text{def}}{=} \mathsf{head}_{\mathsf{L}} \overset{\text{def}}{=} \lambda l.\, \text{if } l = \mathsf{nil} \ \text{ then } d_1 \text{ else } d \text{ where } l = \mathsf{cons}(d, l') \\
\alpha_{\mathsf{FLS}}(\preceq) &\overset{\text{def}}{=} \lambda(l_1, l_2).\, l_1 \preceq_{\mathsf{L}} l_2 \\
\alpha_{\mathsf{FLS}}(\sqcap) &\overset{\text{def}}{=} \lambda(l_1, l_2).\, l_1 \sqcap_{\mathsf{L}} l_2
\end{aligned}$$

Fig. 10. The canonical model α_{FLS} of functional lists with sublists

The theory of functional lists with sublist relationship $\mathcal{T}_{\mathsf{FLS}}$ is the set of all formulas in FLS that are true in the canonical model of lists. We denote this canonical model by α_{FLS}. The structure α_{FLS} is the term algebra generated by the signature $\Sigma_{\mathsf{L}} = (S_{\mathsf{FLS}}, \{\mathsf{cons}, \mathsf{nil}, d_1, d_2, \dots\})$, where $d_1, d_2, \dots$ are infinitely many constant symbols of sort data. The complete definition of α_{FLS} is given in Figure 10. The canonical model interprets the sort list as the set of all Σ_{L}-terms of sort list. We denote this set by L. Likewise, the sort data is interpreted as the set of all Σ_{L}-terms of sort data. We denote this set by D. The function symbols cons and nil are interpreted as the corresponding term constructors. The function symbols head and tail are interpreted as the appropriate selectors $\mathsf{head}_{\mathsf{L}}$ and $\mathsf{tail}_{\mathsf{L}}$. The predicate symbol $\preceq$ is interpreted as the sublist relation $\preceq_L \subseteq \mathsf{L} \times \mathsf{L}$ on lists. The sublist relation is defined as the inverse of the reflexive transitive closure of the tail selector function:

$$l_1 \preceq_{\mathsf{L}} l_2 \overset{\text{def}}{\iff} (l_2, l_1) \in \{\, (l, \mathsf{tail}_{\mathsf{L}}(l)) \mid l \in \mathsf{L} \,\}^*$$

The relation $\preceq_{\mathsf{L}}$ is a partial order on lists. In fact, it induces a meet-semilattice on the set L. We denote by $\sqcap_{\mathsf{L}}$ the meet operator of this semilattice. Given two lists l_1 and l_2, the list $l_1 \sqcap_{\mathsf{L}} l_2$ denotes the greatest common suffix of l_1 and l_2. The structure α_{FLS} interprets the function symbol $\sqcap$ as the operator $\sqcap_{\mathsf{L}}$.

We further define the theory of all finite substructures of α_{FLS}. Let Σ_{FLSf} be the signature Σ_{FLS} without the function symbol cons and let α_{FLSf} be the structure α_{FLS} restricted to the signature Σ_{FLSf}. We call a finite subset L of L sublist closed if for all $l \in L$, $l' \in \mathsf{L}$, $l' \preceq_{\mathsf{L}} l$ implies $l' \in L$. For a finite sublist closed subset L of L, the structure α_L is the finite total substructure of α_{L} induced by the restricted support sets $\alpha_L(\mathsf{list}) \overset{\text{def}}{=} L$ and $\alpha_L(\mathsf{data}) \overset{\text{def}}{=} \{\, \mathsf{head}_{\mathsf{L}}(l) \mid l \in L \,\}$. We denote by $\mathcal{M}_{\mathsf{FLSf}}$ the set of all such finite total substructures α_L of α_{FLSf}. The theory $\mathcal{T}_{\mathsf{FLSf}}$ is the set of all FLS formulas that are true in all structures $\mathcal{M}_{\mathsf{FLSf}}$.

6 Decision Procedure for FLS

In the following, we show that the theory $\mathcal{T}_{\mathsf{FLS}}$ is decidable. For this purpose we reduce the decision problem for $\mathcal{T}_{\mathsf{FLS}}$ to the decision problem of the theory $\mathcal{T}_{\mathsf{FLSf}}$. We then give a finite first-order axiomatization of the theory $\mathcal{T}_{\mathsf{FLSf}}$ and show that it is a local theory extension. In total, this implies that deciding satisfiability of a ground formula F with respect to the theory $\mathcal{T}_{\mathsf{FLS}}$ can be reduced to deciding satisfiability of F conjoined with finitely many ground instances of the first-order axioms of $\mathcal{T}_{\mathsf{FLSf}}$.

Reducing FLS to FLSf. We first note that satisfiability of an FLS formula F in the canonical model can be reduced to checking satisfiability in the finite substructures, if the function symbol cons does not occur in F.

Proposition 1. *Let F be a quantifier-free Σ_{FLSf}-formula. Then F is satisfiable in α_{FLS} if and only if F is satisfiable in some structure $\alpha \in \mathcal{M}_{\mathsf{FLSf}}$.*

We can now exploit the fact that, in the term algebra α_{FLS}, the constructor $\mathsf{cons}_{\mathsf{L}}$ is uniquely determined by the functions $\mathsf{head}_{\mathsf{L}}$ and $\mathsf{tail}_{\mathsf{L}}$. Let F be an FLS formula. Then we can eliminate an occurrence $F(\mathsf{cons}(t_d, t_l))$ of function symbol cons in a term of F by rewriting $F(\mathsf{cons}(t_d, t_l))$ into:

$$x \neq \mathsf{nil} \wedge \mathsf{head}(x) = t_d \wedge \mathsf{tail}(x) = t_l \wedge F(x)$$

where x is a fresh variable of sort list that does not appear elsewhere in F. Let $\mathsf{elimcons}(F)$ be the formula that results from rewriting recursively all appearances of function symbol cons in F. Clearly, in the canonical model α_{FLS}, the formulas F and $\mathsf{elimcons}(F)$ are equisatisfiable. Thus, with Proposition 1 we can conclude.

Lemma 2. *Let F be an FLS formula. Then F is satisfiable in α_{FLS} if and only if $\mathsf{elimcons}(F)$ is satisfiable in some structure $\alpha \in \mathcal{M}_{\mathsf{FLSf}}$.*

Axiomatizing FLSf. We next show that there exists a first-order axiomatization $\mathcal{K}_{\mathsf{FLSf}}$ of the theory $\mathcal{T}_{\mathsf{FLSf}}$. The axioms $\mathcal{K}_{\mathsf{FLSf}}$ are given in Figure 11. The free variables appearing in the formulas are implicitly universally quantified.

Lemma 3. *The axioms $\mathcal{K}_{\mathsf{FLSf}}$ are sound, i.e., for all $\alpha \in \mathcal{M}_{\mathsf{FLSf}}$, $\alpha \models \mathcal{K}_{\mathsf{FLSf}}$.*

Pure: $\mathsf{head}(x) = \mathsf{head}(y) \wedge \mathsf{tail}(x) = \mathsf{tail}(y) \rightarrow x = y \vee x = \mathsf{nil} \vee y = \mathsf{nil}$

NoCycle1:	$\mathsf{nil} \preceq x$	UnfoldL:	$\mathsf{tail}(x) \preceq x$
NoCycle2:	$\mathsf{tail}(x) = x \rightarrow x = \mathsf{nil}$	UnfoldR:	$x \preceq y \rightarrow x = y \vee x \preceq \mathsf{tail}(y)$
Refl:	$x \preceq x$	GCS1:	$x \sqcap y \preceq x$
Trans:	$x \preceq y \wedge y \preceq z \rightarrow x \preceq z$	GCS2:	$x \sqcap y \preceq y$
AntiSym:	$x \preceq y \wedge y \preceq x \rightarrow x = y$	GCS3:	$z \preceq x \wedge z \preceq y \rightarrow z \preceq x \sqcap y$
Total:	$y \preceq x \wedge z \preceq x \rightarrow y \preceq z \vee z \preceq y$		

Fig. 11. First-order axiomatization $\mathcal{K}_{\mathsf{FLSf}}$ of the theory $\mathcal{T}_{\mathsf{FLSf}}$

As a prerequisite for proving completeness of the axioms, we next show that the finite models of the axioms $\mathcal{K}_{\mathsf{FLSf}}$ are structurally equivalent to the finite substructures of the canonical model of functional lists.

Proposition 4. *Every finite model of $\mathcal{K}_{\mathsf{FLSf}}$ is isomorphic to some structure in $\mathcal{M}_{\mathsf{FLSf}}$.*

Locality of FLSf. We will now prove that the theory $\mathcal{K}_{\mathsf{FLSf}}$ can be understood as a local theory extension and, at the same time, prove that $\mathcal{K}_{\mathsf{FLSf}}$ is a complete axiomatization of the theory $\mathcal{T}_{\mathsf{FLSf}}$.

In what follows, the signature Σ_{FLSf} is the signature of the theory extension $\mathcal{K}_{\mathsf{FLSf}}$. We also have to determine the signature Σ_0 of the base theory $\mathcal{T}_0$ by fixing the extension symbols. We treat the function symbols $\Omega_e \overset{\text{def}}{=} \{\mathsf{head}, \mathsf{tail}, \sqcap\}$ as extension symbols, but the sublist relation $\preceq$ as a symbol in the signature of the base theory, i.e. $\Sigma_0 \overset{\text{def}}{=} (S_{\mathsf{FLS}}, \{\mathsf{nil}, \preceq\})$. The base theory itself is given by the axioms that define the sublist relation, but that do not contain any of the extension symbols, i.e., $\mathcal{T}_0 \overset{\text{def}}{=} \{\mathsf{NoCycle1}, \mathsf{Refl}, \mathsf{Trans}, \mathsf{AntiSym}, \mathsf{Total}\}$. We further denote by $\mathcal{K}_e \overset{\text{def}}{=} \mathcal{K}_{\mathsf{FLSf}} \setminus \mathcal{T}_0$ the extension axioms.

We now show that $\mathcal{K}_{\mathsf{FLSf}} = \mathcal{T}_0 \cup \mathcal{K}_e$ is a local theory extension. As in the definition of local theory extensions in Section 4, for a set of ground clauses G, we denote by $\mathcal{K}_e[G]$ all instances of axioms $\mathcal{K}_e$ where the variables occurring below extension symbols Ω_e are instantiated by all ground terms $\mathsf{st}(\mathcal{K}_e, G)$ that appear in $\mathcal{K}_e$ and G. Furthermore, we denote by Σ^c_{FLSf} the signature Σ_{FLSf} extended with finitely many new constant symbols Ω_c.

Lemma 5. *For every finite set of Σ^c_{FLSf} ground clauses G, if α is a partial Σ^c_{FLSf}-structure such that $\alpha_{|\Sigma_0}$ is a total model of $\mathcal{T}_0$, all terms in $\mathsf{st}(\mathcal{K}_e, G)$ are defined in α, and α weakly satisfies $\mathcal{K}_e[G] \cup G$ then there exists a finite total Σ^c_{FLSf}-structure that satisfies $\mathcal{K}_{\mathsf{FLSf}} \cup G$.*

We sketch the proof of Lemma 5. Let α be a partial Σ^c_{FLSf}-structure as required in the lemma. We can obtain a finite partial substructure α' from α by restricting the interpretations of sorts data and list to the elements that are used in the interpretations of the ground terms $\mathsf{st}(\mathcal{K}_e, G)$. Then α' is still a total model of $\mathcal{T}_0$ and still weakly satisfies $\mathcal{K}_e[G] \cup G$, since all axioms in $\mathcal{K}_{\mathsf{FLSf}}$ are universal. We can then complete α' to a finite total model of $\mathcal{K}_{\mathsf{FLSf}} \cup G$ as follows. First, for every $u \in \alpha'(\mathsf{list})$ where $\alpha'(\mathsf{head})$ is not defined, we can extend $\alpha'(\mathsf{data})$ by a fresh element d_u and define $\alpha'(\mathsf{head})(u) = d_u$. Now, let $u \in \alpha'(\mathsf{list})$ such that $\alpha(\mathsf{tail})$ is not defined on u. If $u = \alpha'(\mathsf{nil})$, we define $\alpha'(\mathsf{tail})(u) = u$. Otherwise, from the fact that α' satisfies axioms $\mathsf{NoCycle1}$, $\mathsf{AntiSym}$, and Total we can conclude that there exists a maximal element $v \in \alpha'(\mathsf{list}) \setminus \{u\}$ such that $(v, u) \in \alpha'(\preceq)$. However, we cannot simply define $\alpha'(\mathsf{tail})(u) = v$. The resulting structure would potentially violate axiom Pure. Instead, we extend $\alpha'(\mathsf{list})$ with a fresh element w and $\alpha'(\mathsf{data})$ with a fresh element d_w, and define: $\alpha'(\mathsf{head})(w) = d_w$, $\alpha'(\mathsf{tail})(w) = v$, and $\alpha'(\mathsf{tail})(u) = w$. We further extend the definition of $\alpha'(\preceq)$ for the newly added element w, as expected. The completion of $\alpha'(\sqcap)$ to a total function is then straightforward.

From Lemma 5 we can now immediately conclude that the theory $\mathcal{K}_{\mathsf{FLSf}}$ satisfies condition (Loc). Completeness of the axioms follows from Proposition 4 and Lemma 5.

Theorem 6. *$\mathcal{K}_{\mathsf{FLSf}}$ is a local theory extension of the theory $\mathcal{T}_0$.*

Theorem 7. *$\mathcal{K}_{\mathsf{FLSf}}$ is an axiomatization of the theory $\mathcal{T}_{\mathsf{FLSf}}$, i.e., $\mathcal{T}(\mathcal{K}_{\mathsf{FLSf}}) = \mathcal{T}_{\mathsf{FLSf}}$.*

Deciding FLS. We now describe the decision procedure for deciding satisfiability of FLS formulas. Given an FLS input formula F, the decision procedure proceeds as follows: (1) compute $\hat{F} = \mathsf{elimcons}(\neg F)$, replace all variables in $\hat{F}$ with fresh constant symbols, and transform the resulting formula into a set of ground clauses G; and (2) use Theorem 6 and the reduction scheme for reasoning in local theory extensions [18], to reduce the set of clauses $\mathcal{K}_{\mathsf{FLSf}} \cup G$ to an equisatisfiable formula in the Bernays-Schönfinkel-Ramsey class, which is decidable. The reduction scheme computes the set of clauses $\mathcal{T}_0 \cup \mathcal{K}_e[G] \cup G$ and then eliminates all occurrences of extension functions Ω_e in literals of clauses in this set. The resulting set of clauses contains only universally quantified variables, constants, relation symbols, and equality, i.e., it belongs to the Bernays-Schönfinkel-Ramsey class. Soundness and completeness of the decision procedure follows from Lemma 2, Theorems 6 and 7, and [18, Lemma 4].

Complexity. For formulas in the Bernays-Schönfinkel-Ramsey class that have a bounded number of universal quantifiers, the satisfiability problem is known to be NP-complete [2, page 258]. The only quantified variables appearing in the set of clauses obtained after the reduction step of the decision procedure are those that come from the axioms in $\mathcal{K}_{\mathsf{FLSf}}$, more precisely, the axioms in $\mathcal{T}_0$ and the (partial) instantiations of the axioms in $\mathcal{K}_e$. In fact, we can write the clauses for these axioms in such a way that they use exactly 3 quantified variables. Finally, from the parametric complexity considerations in [18] follows that the size of the set of clauses obtained in the final step of our decision procedure is polynomial in the size of the input formula. It follows that the satisfiability problem for FLS is decidable in NP. NP-hardness follows immediately from the fact that FLS can express arbitrary propositional formulas.

Theorem 8. *The decision problem for the theory $\mathcal{T}_{\mathsf{FLS}}$ is NP-complete.*

7 Extension with Sets of Sublists and Content Sets

We next show decidability of the logic that extends FLS with constraints on sets of sublists and the contents of lists. We do this by reducing the extended logic to constraints on sets. For this we need a normal form of formulas in our logic. To obtain this normal form, we start from partial models of FLS, but refine them further to be able to reduce them to constraints on disjoint sets. We then give a BAPA reduction [24] for each of these refined models.

Predecessor-Refined Partial Structures. Our normal form of an FLS formula F is given by a disjunction of certain partial models α of $\mathcal{K}_{\mathsf{FLSf}}$. We call these models *predecessor-refined partial models.*

Definition 9. *α is a* predecessor-refined partial *(*PRP*)* structure *if it is a partial substructure of a structure in $\mathcal{M}_{\mathsf{FLSf}}$ and the following conditions hold in α*

1. *$\preceq$ is totally defined on $\alpha(\mathsf{list})$*
2. *for all $x, y \in \alpha(\mathsf{list})$, $(x \sqcap y) \in \alpha(\mathsf{list})$. Moreover, if $x, y, (x \sqcap y)$ are three distinct elements, then there exists $x_1 \in \alpha(\mathsf{list})$ such that $x_1 \preceq x$ and $\mathsf{tail}(x_1) = (x \sqcap y)$.*
3. *for all $x, y \in \alpha(\mathsf{list})$, if $x \neq y$ and $\mathsf{tail}(x)$ and $\mathsf{tail}(y)$ are defined and equal, then both $\mathsf{head}(x)$ and $\mathsf{head}(y)$ are defined.*

With each PRP structure α we associate the conjunction of literals that are (strongly) satisfied in α. We call this formula a PRP conjunction.

Theorem 10. *Each FLS formula is equivalent to an existentially quantified finite disjunction of PRP conjunctions.*

We can compute the PRP structures for an FLS formula F by using a simple modification of the decision procedure for FLS presented in Section 6: instead of instantiating the axioms $\mathcal{K}_e$ of the theory extension only with the ground subterms $\mathsf{st}(K_e, G)$ appearing in the extension axioms K_e and the clauses G generated from F, we instantiate the axioms with a larger set of ground terms Ψ defined as follows:

$$\begin{aligned}
\Psi_0 &= \mathsf{st}(K_e, G) \cup \{\, t_1 \sqcap t_2 \mid t_1, t_2 \in \mathsf{st}(K_e, G) \,\} \\
\Psi &= \Psi_0 \cup \{\, \mathsf{head}(t) \mid t \in \Psi_0 \,\} \cup \{\, \mathsf{pre}(t_1, t_2), \mathsf{tail}(\mathsf{pre}(t_1, t_2)) \mid t_1, t_2 \in \Psi_0 \,\}
\end{aligned}$$

Here pre is a fresh binary function symbol, which we introduce as a Skolem function for the existential variable x_1 in Property 2 of PRP structures, i.e., we constrain pre using the following axiom:

$$\mathsf{Pre} : \forall xy.\, x \neq y \land x \neq x \sqcap y \land y \neq x \sqcap y \rightarrow \mathsf{pre}(x, y) \preceq x \land \mathsf{tail}(\mathsf{pre}(x, y)) = x \sqcap y$$

The PRP structures for F are then given by the partial models of $\mathcal{T}_0 \cup (K_e \cup \{\mathsf{Pre}\})[\Psi] \cup G$ in which all terms in Ψ and $\preceq$ are totally defined. These partial models can be computed using a tool such as H-PILoT [5].

Constraints on Sets of Sublists. Define $\sigma(y) = \{x.\ x \preceq y\}$. Our goal is to show that extending FLS with the $\sigma(\_)$ operator and the set algebra of such sets yields in a decidable logic. To this extent, we consider an FLS formula F with free variables $x_1, \ldots, x_n$ and show that the defined relation on sets $\rho = \{(\sigma(x_1), \ldots, \sigma(x_n)).\ F(x_1, \ldots, x_n)\}$ is definable as $\rho = \{(s_1, \ldots, s_n).\ G(s_1, \ldots, s_n)\}$ for some quantifier-free BAPA [8] formula G. By Theorem 10, it suffices to show this property when F is a PRP conjunction, given by some PRP structure α. Figure 12 shows the generation of set constraints from a PRP structure. By replacing each $\sigma(x)$ with a fresh set variable

Input: a PRP structure α. **Output:** a set constraint G_α.

Step 1: Define the relation $\preceq_1$ as irreflexive transitive reduct of $\preceq$ without the tail relation. Formally, for all $x, y \in \alpha(\mathsf{list})$, define $x \preceq_1 y$ iff all of the following conditions hold: (1) $x \preceq y$, (2) $x \neq y$, (3) $\mathsf{tail}(y)$ is undefined, and (4) there is no z in $\alpha(\mathsf{list})$ such that x, y, z are distinct, $x \preceq z$, and $z \preceq y$.

Step 2: Introduce sets $S_{x,y}$ with the meaning $S_{x,y} = (\sigma(y) \setminus \sigma(x)) \setminus \{y\}$ and define $\mathsf{Segs} = \{S_{x,y} \mid x \preceq_1 y\}$.

Step 3: Generate the conjunction $\hat{G}_\alpha$ of the following constraints:

1. $\sigma(\mathsf{nil}) = \{\mathsf{nil}\}$
2. $\sigma(y) = \{y\} \cup \sigma(x)$, for each x, y such that α satisfies $\mathsf{tail}(y) = x$
3. $\sigma(y) = \{y\} \cup S_{x,y} \cup \sigma(x)$, for each x, y such that α satisfies $x \preceq_1 y$
4. $\mathsf{disjoint}((S)_{S \in \mathsf{Segs}}, (\{x\})_{x \in \alpha(\mathsf{list})})$

Step 4: Existentially quantify over all Segs variables in $\hat{G}_\alpha$. If the goal is to obtain a formula without Segs variables, replace each variable $S_{x,y}$ with $(\sigma(y) \setminus \sigma(x)) \setminus \{y\}$.

Step 5: Return the resulting formula G_α.

Fig. 12. Generation of set constraints from a PRP structure

s_x in the resulting constraint we obtain a formula in set algebra. We can check the satisfiability of such formulas following the algorithms in [8].

Among the consequences of this reduction is NP-completeness of a logic containing atomic formulas of FLS, along with formulas $s = \sigma(x)$, set algebra expressions containing $\subseteq, \cap, \cup, \setminus, =$ on sets, and the cardinality operator $\mathsf{card}(s)$ that computes the size of the set s along with integer linear arithmetic constraints on such sizes. Because the length of the list x is equal to $\mathsf{card}(\sigma(x)) - 1$, this logic also naturally supports reasoning about list lengths. We note that such a logic can also support a large class of set comprehensions of the form $S = \{x. F(x, y_1, \ldots, y_n)\}$ when the atomic formulas within F are of the form $u \preceq v$ and at least one atomic formula of the form $x \preceq y_i$ occurs positively in disjunctive normal form of F. Because $\forall x.F$ is equivalent to $\mathsf{card}(\{x. \neg F\}) = 0$, sets give us a form of universal quantification on top of FLS.

Additional Constraints on List Content. We next extend the language of formulas to allow set constraints not only on the set of sublists $\sigma(x)$ but also on the images of such sets under the head function. We define the list content function by $\tau(x) = \mathsf{head}[\sigma(x) \setminus \{\mathsf{nil}\}]$ where we define $\mathsf{head}[s] = \{\mathsf{head}(x) \mid x \in s\}$. We then obtain our full logic FLS^2 shown in Figure 8 that introduces constraints of the form $\mathsf{head}[s] = v$ on top of FLS and constraints on sets of sublists. To show decidability of this logic, we use techniques inspired by [25] to eliminate the image constraints. The elimination procedure is shown in Figure 13. We use the properties of PRP structures that the elements for which $\mathsf{tail}(x_L) = \mathsf{tail}(x_R)$ holds have defined values $\mathsf{head}(x_L)$ and $\mathsf{head}(x_R)$. This allows us to enforce sufficient conditions on sets of sublists and sets of their heads to ensure that the axiom Pure can be enforced. The elimination procedure assumes that we have $\mathsf{head}(s)$ expressions only in the cases where s is a combination of sets of the

Input: a PRP structure α and an image constraint C.
Output: a set constraint C_α without $\mathsf{head}[s]$ expressions
Step 1: Replace each $\tau(x)$ in C with $\mathsf{head}[\sigma(x) \setminus \{\mathsf{nil}\}]$.
Step 2: Let P_i be all sets of the form $\{x_i\}$ or S_{x_i,x_j} from Figure 12. If s is a Boolean combination of expressions of the form $\sigma(x)$, $\{x\}$, let $J(s)$ be such that $s = \bigcup_{i \in J(s)} P_i$ is the decomposition of s into disjoint sets, derived from set equalities in Figure 12. Then replace each expression $\mathsf{head}[s]$ with $\bigcup_{i \in J(s)} \mathsf{head}[P_i]$.
Step 3: Replace each $\mathsf{head}[P_i]$ with a fresh set variable Q_i and conjoin the result with the following constraints on the image sets Q_i:
1. $\mathsf{card}(Q_i) \leq \mathsf{card}(P_i)$
2. $Q_i = \emptyset \rightarrow P_i = \emptyset$
3. $Q_i \cap Q_j = \emptyset$, for each $x, y \in \alpha(\mathsf{list})$ such that $P_i = \{x\}$, $P_j = \{y\}$, $x \neq y$ and $\mathsf{tail}(x) = \mathsf{tail}(y)$.

Step 4: Existentially quantify over all Q_i and return the resulting formula C_α.

Fig. 13. Eliminating $\mathsf{head}[s]$ from image constraints by introducing additional constraints on top of Figure 12

form $\sigma(x)$ and $\{x\}$, which ensures that s is a disjoint combination of polynomially many partitions. This restriction is not necessary [25], but is natural in applications and ensures the membership in NP.

8 Conclusion

We presented a new decidable logic that can express interesting properties of functional lists and has a reasonably efficient decision procedure. We showed that this decision procedure can be useful to increase the degree of automation in verification tools and interactive theorem provers.

References

1. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. ENTCS 174(8), 23–37 (2007)
2. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer, Heidelberg (1997)
3. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: A Logic-Based Framework for Reasoning about Composite Data Structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
4. Furia, C.A.: What's Decidable about Sequences? In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 128–142. Springer, Heidelberg (2010)
5. Ihlemann, C., Sofronie-Stokkermans, V.: System Description: H-PILoT. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 131–139. Springer, Heidelberg (2009)
6. Jacobs, S.: Incremental Instance Generation in Local Reasoning. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 368–382. Springer, Heidelberg (2009)

7. Jaffar, J.: Minimal and complete word unification. J. ACM 37(1), 47–85 (1990)
8. Kuncak, V., Rinard, M.: Towards Efficient Satisfiability Checking for Boolean Algebra with Presburger Arithmetic. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 215–230. Springer, Heidelberg (2007)
9. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL (2008)
10. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating Reachability using First-Order Logic with Applications to Verification of Linked Data Structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
11. Makanin, G.: The problem of solvability of equations in a free semigroup. Math. USSR Sbornik, 129–198 (1977); AMS (1979)
12. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated Verification of Shape, Size and Bag Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
13. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
14. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)
15. Oppen, D.C.: Reasoning about recursively defined data structures. In: POPL, pp. 151–157 (1978)
16. Piskac, R., Suter, P., Kuncak, V.: On decision procedures for ordered collections. Technical Report LARA-REPORT-2010-001, EPFL (2010)
17. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM 51(3) (2004)
18. Sofronie-Stokkermans, V.: Hierarchic Reasoning in Local Theory Extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
19. Sofronie-Stokkermans, V.: Locality Results for Certain Extensions of Theories with Bridging Functions. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 67–83. Springer, Heidelberg (2009)
20. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
21. Venkataraman, K.N.: Decidability of the purely existential fragment of the theory of term algebras. Journal of the ACM (JACM) 34(2), 492–510 (1987)
22. Wies, T., Muñiz, M., Kuncak, V.: On deciding functional lists with sublist sets. Technical Report EPFL-REPORT-148361, EPFL (2010), `http://cs.nyu.edu/~wies/publ/on_deciding_functional_lists_with_sublist_sets.pdf`
23. Wies, T., Muñiz, M., Kuncak, V.: An Efficient Decision Procedure for Imperative Tree Data Structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011)
24. Wies, T., Piskac, R., Kuncak, V.: Combining Theories with Shared Set Operations. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 263–278. Springer, Heidelberg (2009)
25. Yessenov, K., Kuncak, V., Piskac, R.: Collections, Cardinalities, and Relations. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 380–395. Springer, Heidelberg (2010)
26. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI (2008)

Developing Verified Programs with Dafny

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. Dafny [2] is a programming language and program verifier. The language is type-safe and sequential, and it includes common imperative features, dynamic object allocation, and inductive datatypes. It also includes specification constructs like pre- and postconditions, which let a programmer record the intended behavior of the program along with the executable code that is supposed to cause that behavior. Because the Dafny verifier runs continuously in the background, the consistency of a program and its specifications is always enforced.

Dafny has been used to verify a number of challenging algorithms, including Schorr-Waite graph marking, Floyd's "tortoise and hare" cycle-detection algorithm, and snapshotable trees with iterators. Dafny is also being used in teaching and it was a popular choice in the VSTTE 2012 program verification competition. Its open-source implementation has also been used as a foundation for other verification tools.

In this tutorial, I will give a taste of how to use Dafny in program development. This will include an overview of Dafny, basics of writing specifications, how to debug verification attempts, how to formulate and prove lemmas, and some newer features for staged program development.

References

1. Leino, K.R.M.: Specification and verification of object-oriented software. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press (2009); Summer School Marktoberdorf 2008 Lecture Notes
2. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
3. Leino, K.R.M.: Automating induction with an SMT solver. In: VMCAI (to appear, 2012)

Verifying Two Lines of C with Why3: An Exercise in Program Verification[⋆]

Jean-Christophe Filliâtre

CNRS
LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay-Île-de-France, ProVal, Orsay F-91893

Abstract. This article details the formal verification of a 2-line C program that computes the number of solutions to the n-queens problem. The formal proof of (an abstraction of) the C code is performed using the Why3 tool to generate the verification conditions and several provers (Alt-Ergo, CVC3, Coq) to discharge them. The main purpose of this article is to illustrate the use of Why3 in verifying an algorithmically complex program.

1 Introduction

Even the shortest program can be a challenge for formal verification. This paper exemplifies this claim with the following 2-line C program:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

This rather obfuscated code was found on a web page gathering C signature programs[1] and was apparently authored by Marcel van Kervinc. This is a standalone C program that reads an integer n from standard input and prints another integer $f(n)$ on standard output. If n is smaller than the machine word size in bits (typically 32), then $f(n)$ appears to be the number of solutions to the well-known n-queens problem, that is the number of ways that n queens can be put on a $n \times n$ chessboard so that they do not attack each other. More surprisingly, this is a very efficient program to compute this number.

As a case study for Why3, a tool the author of this paper is co-developing [4], we consider verifying this program formally. Since Why3 is not addressing C programs, we make an abstraction of the algorithm above. Our goal is then a mechanically-assisted proof that this algorithm terminates and indeed computes the expected number. This is highly challenging, due to the algorithmic complexity of this program. The main contribution of this paper is to demonstrate the ability of our tool to tackle the wide range of verification issues involved in such a proof. In particular, it shows the relevance of using both automated and

⋆ This work is partly supported by the ANR project *U3CAT* (ANR-08-SEGI-021) and the Open-DO project Hi-Lite.

[1] http://www.iwriteiam.nl/SigProgC.html

interactive theorem provers within the same framework. Additionally, this paper provides a nice benchmark for people developing tools for the verification of C programs; they may consider refining our proof into a proof of the C code above.

This paper is organized as follows. Section 2 "unobfuscates" the program, explaining the algorithm and its data. Section 3 briefly introduces Why3, a tool which takes annotated code as input and produces verification conditions in the native syntax of several existing provers. Section 4 details the verification process, namely the logical annotations inserted in the program and the methods used to discharge the resulting verification conditions. We conclude with a discussion in Section 5. Annotated source code and proofs are available online at `http://why3.lri.fr/queens/`. Proofs can be replayed in a batch mode.

2 Unobfuscation

Before we enter the formal verification process, we first explain this obfuscated C program. The code is divided into a recursive function `t`, which takes three integers as arguments and returns an integer, and a main function which reads an integer from standard input, calls function `t` and prints the result on standard output. With added type declarations and a bit of indentation, function `t` reads as follows:

```
int t(int a, int b, int c) {
  int d=0,e=a&~b&~c,f=1;
  if(a) for(f=0; d=(e-=d)&-e; f+=t(a-d,(b+d)*2,(c+d)/2));
  return f;
}
```

The assignment `d=(e-=d)&-e` does not strictly conform with ANSI C standard, because it assumes that the inner assignment `e-=d` is performed before evaluating `-e`. This is not guaranteed and the compiler may freely choose between both possible evaluation strategies. It is easy to turn the code in legal C: since `d` is initialized to 0, we can safely move assignment `e-=d` to the end of the loop body. Then we do not need the initialization `d=0` anymore[2]. The second modification we make is to replace the `main` function with a `queens` function from `int` to `int`, since we are only interested in the integer function and not in input-outputs. We end up with the code given in Fig. 1. Our goal is to show that `queens`(n) is indeed the number of solutions to the n-queens problem.

Let us now explain the algorithm and its data. This is a backtracking algorithm which fills the rows of the chessboard one at a time. More precisely, each call to `t` enumerates all possible positions for a queen on the current row inside the `for` loop and, for each of them, recursively calls `t` to fill the remaining rows. The number of solutions is accumulated in `f` and returned. The key idea is to use integers as *sets* or, equivalently, as *bit vectors*: i belongs to the "set" x if and only if the i-th bit of x is set. According to this trick, program variables `a`, `b`, `c`,

[2] This even reduces the size of the original code.

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d,(b+d)*2,(c+d)/2));
  return f;
}
int queens(int n) {
  return t(~(~0<<n),0,0);
}
```

Fig. 1. Unobfuscated C code

```
int $t$(set $a$, set $b$, set $c$)
    $f \leftarrow 1$
    if $a \neq \emptyset$
        $e \leftarrow (a \setminus b) \setminus c$
        $f \leftarrow 0$
        while $e \neq \emptyset$
            $d \leftarrow min\_elt(e)$
            $f \leftarrow f + t(a \setminus \{d\}, succ(b \cup \{d\}), pred(c \cup \{d\}))$
            $e \leftarrow e \setminus \{d\}$
    return $f$

int $queens$(int $n$)
    return $t(\{0, 1, \ldots, n-1\}, \emptyset, \emptyset)$
```

Fig. 2. Abstract version of the code using sets

`d` and `e` are seen as subsets of $\{0, 1, \ldots, n-1\}$. Then almost all computations in this program are to be understood as set operations. Some of them are clear: `a&~b&~c` computes the set `a` $\setminus$ `b` $\setminus$ `c`, the test `if(a)` checks whether `a` is empty, etc. Others are more subtle. For instance, `e&-e` computes the smallest element of `e` (and returns the corresponding singleton set). This is a nice property of the two's complement arithmetic; see for instance [14,10] for an explanation[3]. Then the result `d` can be removed from set `a` using subtraction `a-d` since the bit of `d` that is set is also set in `a`; similarly, `d` is added to sets `b` and `c` using a mere addition since the corresponding bit is not set in `b` and `c`. Another trick is the computation of the set $\{0, 1, \ldots, \texttt{n}-1\}$ as `~(~0<<n)`. Finally, multiplication by 2 (resp. division by 2) is used to add 1 (resp. subtract 1) to each element of a set; from now on, we use *succ* and *pred* to denote those two set operations. We can now write a more abstract version of the code that only deals with finite sets. It is given in Fig. 2. Note that n, f, and returned values of t and *queens* are still integers.

It is now easier to explain the algorithm. Set a contains the columns not yet assigned to a queen, *i.e.* candidate positions for the queen to be set on the current

[3] This trick is used in Patricia trees [11] implementations.

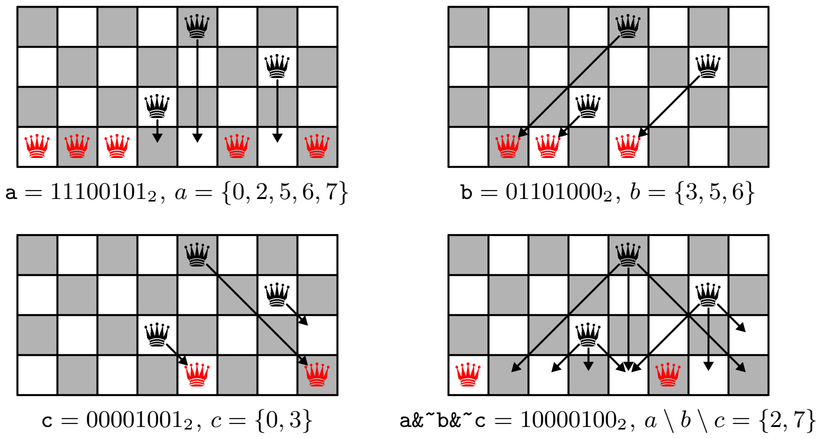
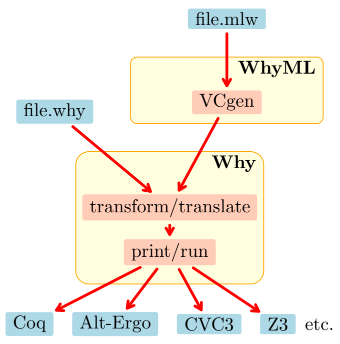

$\mathtt{a} = 11100101_2,\ a = \{0, 2, 5, 6, 7\}$
$\mathtt{b} = 01101000_2,\ b = \{3, 5, 6\}$
$\mathtt{c} = 00001001_2,\ c = \{0, 3\}$
$\mathtt{a\&\tilde{}b\&\tilde{}c} = 10000100_2,\ a \setminus b \setminus c = \{2, 7\}$

Fig. 3. Interpretation of variables `a`, `b`, and `c` as sets

row. Initially, a contains all possible positions, that is $a = \{0, 1, \ldots, n-1\}$. If we have found one solution, a becomes empty, then we return 1. Otherwise, we have to consider all possible positions on the current row. Sets b and c respectively contain the positions to be avoided because they are on an ascending (resp. descending) diagonal of a queen on previous rows. Thus $e = a \setminus b \setminus c$ precisely contains the positions to be considered for the current row. They are all examined one at a time by repeatedly removing the smallest element from e, which is set to d. Then next rows are considered by a recursive call to t with a, b and c being updated according to the choice of column d for the current row: d is removed from the set of possible columns ($a \setminus \{d\}$), added to the set of ascending diagonals which is shifted ($succ(b \cup \{d\})$, and similarly added to the set of descending diagonals which is shifted the other way ($pred(c \cup \{d\})$). The values of a, b and c are illustrated in Fig. 3 for $n = 8$ on a configuration where 3 rows are already set (columns are numbered right to left, starting from 0).

3 Overview of Why3

Why3 is a set of tools for program verification. Basically, it is composed of two parts, which are depicted to the right: a logical language called Why with an infrastructure to translate it to existing theorem provers; and a programming language called WhyML with a verification condition generator.



The logic of Why3 is a polymorphic first-order logic with algebraic data types and inductive predicates [5]. Logical declarations are organized in small units called *theories*. In the following, we use two such theories from

```
let rec t (a b c: set int) =
  if not (is_empty a) then begin
    let e = ref (diff (diff a b) c) in
    let f = ref 0 in
    while not (is_empty !e) do
      let d = min_elt !e in
      f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
      e := remove d !e
    done;
    !f
  end else
    1

let queens (q: int) =
  t (below q) empty empty
```

Fig. 4. Why3 code for the program in Fig. 2

Why3's standard library: integers and finite sets of integers. The latter provides a type `set int` and several operations: a constant `empty` for the empty set; functions `add`, `remove`, `diff`, `min_elt`, `cardinal`, and `below` (`below` n is the set $\{0, 1, \ldots, n-1\}$); a predicate `is_empty`. Operations `succ` and `pred` are missing from this library and we need to introduce them. First, we declare them as follows:

```
function succ (set int) : set int
function pred (set int) : set int
```

Then we axiomatize them as follows:

```
axiom succ_def:
  ∀ s: set int, i: int. mem i (succ s) ↔ i ≥ 1 ∧ mem (i-1) s
axiom pred_def:
  ∀ s: set int, i: int. mem i (pred s) ↔ i ≥ 0 ∧ mem (i+1) s
```

Why3 provides a way to show the consistency of these axioms (by providing a definition in Coq); however, we haven't done it.

On top of this logic, Why3 provides a programming language, WhyML, with a verification condition generator. This is a first-order language with an ML-flavored syntax. It provides the usual constructs of imperative programming (while loop, sequence, exceptions) as well as several constructs of ML (pattern matching, local functions, polymorphism). All symbols from the logic (types, functions, predicates) can be used in programs. Mutable data types can also be introduced, by means of record types with mutable fields. This includes polymorphic references, which are part of Why3's standard library. A reference `r` to a value of type τ has type `ref` τ, is created with function `ref`, is accessed with `!r`, and assigned with `r := `e. Why3 code for the program in Fig. 2 is given in Fig. 4.

Programs are annotated using pre- and postconditions, loop invariants, and variants to ensure termination. Verification conditions are computed using a weakest precondition calculus and then passed to the back-end of Why3 to be sent to theorem provers.

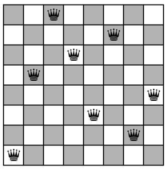
4 Verification

We focus here on the verification of the code in Fig. 4. (The verification of the original C code in Fig. 1 is discussed at the end of this paper.) We need to prove three properties regarding this code: it does not fail, it terminates, and it indeed computes the number of solutions to the n-queens problem. The first property is immediate since there is no division, no array access, or any similar operation that could fail. We will consider termination later, as part of the verification process (Sec. 4.2). Let us first focus on the specification.

4.1 Specification

We need to express that the value returned by a call to `queens` n is indeed the number of solutions to the n-queens problem. As we have seen, the program is building solutions one by one. Thus we have to prove that it finds *all* solutions, *only* solutions and that it does *not find* the same solution *twice*. There is a major difficulty here: the program is not storing anything, not even the current solution being built. How can we state properties about the solutions being found?

One solution is to use *ghost code*, that is additional code not participating in the computation of the final result but potentially accessing the program data. This ghost code will fill an array with all solutions. One solution is represented by an array of n integers. Each cell gives the column assigned to the queen on the corresponding row. For instance, the array $\boxed{5}\boxed{2}\boxed{4}\boxed{6}\boxed{0}\boxed{3}\boxed{1}\boxed{7}$ corresponds to the solution of the 8-queens problem displayed to the right. Rows are numbered from top to bottom and columns from right to left — the latter follows the usual convention of displaying least significant bits to the right, as in Fig. 3. Arrays used in ghost code do not really have to be "true" arrays: there is need neither for efficiency, nor for array bound checking. Thus we can model such arrays using purely applicative maps from Why3's standard library. Thus we simply define



```
type solution = map int int
```

We introduce a global variable `col` to record the current solution under construction, as well as a global variable `k` to record the next row to be filled:

```
val col: ref solution   (* solution under construction *)
val k  : ref int        (* next row in the current solution *)
```

The set of all solutions found so far is recorded into another array. It has type

```
type solutions = map int solution
```

and is declared as a global variable `sol`, together with another global variable `s` holding the next empty slot in `sol`:

```
val sol: ref solutions (* all solutions *)
val s  : ref int       (* next slot for a solution *)
```

If solutions are stored in sol starting from index 0, then s is also the total number of solutions found so far.

Using these four ghost variables, we can instrument the program with ghost code to record the solutions. First, we surround the recursive call to f (line 7) with code to record the value d for row k and to update k:

```
 (* ghost *) col := !col[!k ← d];
 (* ghost *) incr k;
 f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
 (* ghost *) decr k;
```

Function incr (resp. decr) is a shortcut to increase (resp. decrease) an integer reference. Second, when a solution is found, we record it into sol and increase s by one, just before returning 1 (line 12):

```
  else begin
    (* ghost *) sol := !sol[!s ← !col];
    (* ghost *) incr s;
    1
  end
```

So far we have instrumented the code to record the solutions it finds. We still have to define what a solution is and to use this definition to specify the code. From now on, it is convenient to introduce the number n of queens as a parameter:

```
function n : int
```

This is not a limitation: a suitable precondition to function queens will say that its argument q is equal to n (and we don't even have callers). An alternative would be to pass n as a parameter everywhere, but we prefer avoiding it for greater clarity. To define what a solution is, we first define the notion of *partial solution*, up to row k (excluded):

```
predicate partial_solution (k: int) (s: solution) =
  ∀ i: int. 0 ≤ i < k →
  0 ≤ s[i] < n ∧
  (∀ j: int. 0 ≤ j < i →
     s[i] ≠ s[j] ∧ s[i]-s[j] ≠ i-j ∧ s[i]-s[j] ≠ j-i)
```

Note that we avoid the use of the absolute value function: we do so to relieve the automated theorem provers from resorting to the definition (typically an axiom). The notion of solution is derived immediately, by instantiating k with n:

```
predicate solution (s: solution) = partial_solution n s
```

To prove the absence of duplicate solutions, it is convenient to equip the set of solutions with a *total order*. It is naturally given by the code: since elements of e are processed in increasing order, by repeated use of function min_elt, solutions are found in lexicographic order.

For instance, the first five solutions for $n = 8$ are displayed to the right. To define the lexicographic order, we first define the property for two arrays to have a common prefix of length `i`:

0	4	7	5	2	6	1	3
0	5	7	2	6	3	1	4
0	6	3	5	7	1	4	2
0	6	4	7	1	3	5	2
1	3	5	7	2	0	6	4
⋮							

```
predicate eq_prefix (t u: map int α) (i: int) =
  ∀ k: int. 0 ≤ k < i → t[k] = u[k]
```

We make this a polymorphic predicate, to reuse it on both solutions and arrays of solutions. Then it is easy to define the lexicographic order over solutions:

```
predicate lt_sol (s1 s2: solution) =
  ∃ i: int. 0 ≤ i < n ∧ eq_prefix s1 s2 i ∧ s1[i] < s2[i]
```

Finally, we introduce two convenient shortcuts for the forthcoming specifications. Equality of two solutions is defined using `eq_prefix`:

```
predicate eq_sol (t u: solution) = eq_prefix t u n
```

The property for an array of solutions `s` to be sorted in increasing order between index `a` included and index `b` excluded is defined in an obvious way:

```
predicate sorted (s: solutions) (a b: int) =
  ∀ i j: int. a ≤ i < j < b → lt_sol s[i] s[j]
```

This completes the set of definitions needed to specify the code's behavior. The full specification for function `queens` (lines 14–15) is the following[4]:

```
let queens (q: int) =
  { 0 ≤ q = n ∧ !s = 0 ∧ !k = 0 }
  t (below q) empty empty
  { result = !s ∧ sorted !sol 0 !s ∧                          (S)
    ∀ u: solution.
    solution u ↔ (∃ i:int. 0 ≤ i < result ∧ eq_sol u !sol[i]) }
```

The precondition requires both `s` and `k` to be initially equal to zero. The postcondition states that the returned value is equal to the number of solutions stored in array `sol`, that is `!s`. Additionally, it states that array `sol` is sorted and that an array `u` is a solution if and only if it appears in `sol`.

At this point, the reader should be convinced that specification (S) is indeed expressing that this program is computing the number of solutions to the n-queens problem. This is slightly subtle, since the absence of duplicated solutions is not immediate: it is only a provable consequence of `sol` being sorted. Our proof includes this property as a lemma.

4.2 Correctness Proof

We now have to prove that function `queens` terminates and obeys specification (S) above. As a warm-up, let us prove termination first.

[4] The code with all annotations is given in the appendix.

Termination. Termination reduces to that of function `t`. This involves proving its termination as a recursive function, as well as proving the termination of the `while` loop it contains. The termination of the `while` loop (lines 5–9) is immediate, since the cardinality of `e` is decreased by one at each step of the loop. We give the loop a variant accordingly:

```
while not (is_empty !e) do variant { cardinal !e } ...                 (V1)
```

The proof is immediate. Regarding the termination of recursive calls, there is also an obvious variant, namely the cardinality of `a`. It is indeed decreased by one at each recursive call. We give this variant for function `t` as follows:

```
let rec t (a b c: set int) variant { cardinal a } = ...                (V2)
```

The proof is not immediate, however. Indeed, for the cardinality to decrease, we have to prove that `d` is an element of `a`. Within the loop, we only know for sure that `d` is an element of `e`. Thus we need a loop invariant to maintain that `e` is included in `a`. This could be the following:

```
while not (is_empty !e) do invariant { subset !e a } ...
```

However, we will later need a more accurate invariant, which states that `e` remains included in its initial value, that is `diff (diff a b) c`. Thus we favor the following invariant:

```
while not (is_empty !e) do
  invariant { subset !e (diff (diff a b) c) } ...                      (I1)
```

Remaining Annotations. To prove that function `queens` satisfies specification (S) above, we have to give function `t` a suitable specification as well. Obviously, this is a generalization of specification (S). Let us start with the precondition for `t`. First, variable `k` must contain a valid row number and `s` should be non-negative:

```
{ 0 ≤ !k ∧ !k + cardinal a = n ∧ !s ≥ 0 ∧ ... }                        (P1)
```

Second, sets `a`, `b`, and `c` must contain elements that are consistent with the contents of array `col`:

```
{ ...
  (∀ i: int. mem i a ↔
    (0 ≤ i < n ∧ ∀ j: int. 0 ≤ j < !k →  !col[j] ≠ i)) ∧
  (∀ i: int. i ≥ 0 → not (mem i b) ↔                                   (P2)
    (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + j - !k)) ∧
  (∀ i: int. i ≥ 0 → not (mem i c) ↔
    (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + !k - j)) ∧ ... }
```

Finally, array `col` must contain a partial solution up to row `k` excluded:

```
{ ... partial_solution !k !col }                                       (P3)
```

This completes the precondition for function `t`. Let us consider now its postcondition. First, it says that `s` must not decrease and that `k` must not be modified:

```
{ result = !s - old !s ≥ 0 ∧ !k = old !k ∧ ... }                       (Q1)
```

Then it says that all solutions found in this run of `t`, that is between the initial and final values of `s`, must be sorted in increasing order:

```
{ ... sorted !sol (old !s) !s ∧ ... }                                  (Q2)
```

Additionally, these new solutions must be exactly the solutions extending the first `k` rows of array `col`:

```
{ ...
  (∀ u: solution.
     solution u ∧ eq_prefix !col u !k ↔                                (Q3)
     ∃ i: int. old !s ≤ i < !s ∧ eq_sol u !sol[i]) ∧ ... }
```

Finally, the first `k` rows of `col` must not be modified, and so are the solutions that were contained in `sol` prior to the call to `t`:

```
{ ... eq_prefix (old !col) !col !k ∧
      eq_prefix (old !sol) !sol (old !s) }                             (Q4)
```

With such pre- and postcondition for function `t`, function `queens` can be proved correct easily (verification conditions are discharged automatically).

The last step in the specification process is to come up with a loop invariant for function `t` (lines 5–9). It should be strong enough to establish postconditions (Q_1)–(Q_4). We already came up with invariant (I_1) to ensure termination. To ensure postcondition (Q_1), there is an obvious invariant regarding `s` and `k`:

```
{ ... !f = !s - at !s 'L ≥ 0 ∧ !k = at !k 'L ∧ ... }                  (I2)
```

Notation `at !s 'L` is used to refer to the value of `s` at the program point designated by label `'L`. This label is introduced before the `while` keyword at line 5 (this label appears in the code given in the appendix).

One key property to ensure that solutions are found in increasing order for `lt_sol` is that we traverse elements of `e` in increasing order, by repeated extraction of its minimum element. This must be turned into a loop invariant. It states that elements of `e` already considered are all smaller than elements of `e` yet to be considered:

```
{ ...
  (∀ i j: int.                                                         (I3)
    mem i (diff (at !e 'L) !e) → mem j !e → i < j) ∧ ... }
```

Additionally, we must maintain that solutions found in this run of `t`, that is between the initial value of `s` and its current value, are sorted in increasing order:

```
{ ... sorted !sol (at !s 'L) !s ∧ ... }                                (I4)
```

We also have to maintain property (P_3), since array `col` is modified by recursive calls to `t`:

```
{ ... partial_solution !k !col ∧ ... }                                 (I5)
```

The most complex part of the loop invariant is surely the following, which is needed to ensure postcondition (Q_3). It states that the solutions found so far in this run of function `t` are exactly those extending the first `k` rows of `col` with an element of `e` already processed:

```
{ ...
  (∀ u: solution.
    solution u ∧ eq_prefix !col u !k ∧
    mem u[!k] (diff (at !e 'L) !e)                              (I6)
    ↔
    ∃ i: int. (at !s 'L) ≤ i < !s ∧ eq_sol u !sol[i]) ∧ ... }
```

Finally, we complete the loop invariant with an invariance property for `col` and `sol` similar to (Q_4):

```
{ ... eq_prefix (at !col 'L) !col !k ∧                            (I7)
      eq_prefix (at !sol 'L) !sol (at !s 'L) }
```

This completes the specification for function `t`. Fully annotated code is given in the appendix. We end up with 46 lines of annotations (not including the preliminary definitions and axiomatizations!) for 2 lines of code. This huge ratio should be considered as extreme: we are proving a very complex property of a smart algorithm.

Mechanical Proof. The proof is performed using the SMT solvers Alt-Ergo [3] and CVC3 [1], and the Coq proof assistant [13,2]. Running Why3 on the resulting annotated source code produces 41 verification conditions for function `t` and 2 for function `queens`. The latter are automatically discharged by CVC3. As expected, verification conditions for `t` are more difficult to prove. Only 35 of them are discharged automatically, either by Alt-Ergo or CVC3. The remaining 6 verification conditions are discharged manually, using the Coq proof assistant. They are the following:

– precondition (P_2) for the recursive call to `t` (3 goals, corresponding to the 3 right to left implications);
– preservation of invariant (I_3);
– preservation of invariant (I_4);
– postcondition (Q_3) (left to right implication).

The Coq proof scripts amount to 142 lines of tactics and represent a few hours of work. It is important to point out that these Coq proofs only involve steps that could, in principle, be performed by SMT solvers as well (case analysis, Presburger arithmetic, definition expansion, rewriting, quantifier instantiation).

Beside verification conditions, our proof also contains two lemmas: one for the absence of duplicate solutions (see end of Section 4.1) and one technical lemma regarding `partial_solution`. They are respectively discharged by CVC3 and Alt-Ergo.

5 Discussion

We have presented the formal verification of an extremely short but also extremely complex program using Why3. Beyond being a nice specification exercise, it was the opportunity to introduce program verification using Why3 and to illustrate several key features such as user axiomatizations or combined use of interactive and automated theorem provers. We conclude this paper with several discussions.

Originality. The verification competition organized during VSTTE 2010 [9] already included a problem related to the n-queens problem. It was simpler, though, since the code to be verified only had to check the existence of at least one solution (and to return one, if any).

Ghost Code. This case study is yet another example of where *ghost code* is useful in verification [12]. In this particular case, the program is enumerating the solutions to a problem, but does not store any of them, not even the current one. Thus we enriched the code with new statements so that a rich specification is possible. There is currently no support for ghost code in Why3; we plan to add this feature in the future. In particular, this will include a check that (1) ghost code is not modifying the program data, and (2) the program is not accessing the ghost data. In this proof, we have only performed this verification manually.

Verification of the Original C Code. We have not verified the original C code, only its abstraction into WhyML. Regarding the code structure, this is not really an issue, since all C features involved (recursive function, while loop, mutable variables) are available in WhyML as well. Regarding the code data, on the contrary, our proof did not consider the use of integers as bit vectors; we used sets instead. Our purpose was to focus on the specification of the algorithm.

Now that we have come up with a suitable specification, we could refine our proof into a proof of the original C code. A possible route is to introduce a function symbol, say `bits`, that turns an integer into the set of 1-bits in its two's complement representation. Then we can mechanically translate all the annotations, replacing `a` with `bits a`, `b` with `bits b`, and so on. The only change in the annotations is likely to be an extra precondition stating that the upper bits of `c` are zeros (otherwise, ones could be erroneously introduced by the divisions by two). The proof then requires extra lemmas to justify the tricks used in the code. For instance, a lemma will show that, under suitable conditions on `x`, we have `bits (x & -x) = singleton (min_elt (bits x))`. A bit vector library with two's complement interpretations is currently under development in Why3; we consider refining our proof along the lines we just sketched in a future work. Last, translating the resulting proof into a verification tool for C programs, such as VCC [6] or Frama-C [8], should be straightforward. It would be interesting to see which level of proof automation can be achieved.

Overflows. There are two kinds of integer overflows in this program, depending on the use of integers as bit vectors or as counters. Regarding integers used as bit vectors, we can easily cope with the boundedness of integers by imposing the precondition $n \leq$ `size` where `size` stands for the machine word size[5]. The program is performing overflows as soon as $n >$ `size`$/2$ since `b` may contain bits which will overflow due to the repetitive multiplications by 2. These are harmless overflows, but any suitable model should allow them.

Yet there is another source of integer overflows, in variable `f` and the returned value[6]. And it is more difficult to cope with. Even unsigned, 32-bit integers are not large enough to hold the number of solutions to the n-queens problem as soon as $n \geq 19$, the number of solutions for $n = 19$ being 4,968,057,848. Even if we use 64-bit integers for the result, we would need to limit n accordingly (most likely with $n \leq 28$) and then to prove the absence of overflow. But this would in turn require to know the number of solutions, which is precisely what we are trying to compute. An upper bound for the number of solutions would be enough, but there is no good one (and even if it would exist, this would require to be proved). One workaround would be to make the code detect overflows, and fail in such a case. Then our proof can be seen as a proof of the following statement: "if there is no overflow, then the returned value is indeed the number of solutions". Another workaround would be to perform the computation using arbitrary precision integers, which would be faithful to the proof we have made. But this would slow the computation; considering that a record attempt already requires dozens of years of total CPU time, we can hardly afford slowing it.

Acknowledgement. Natarajan Shankar kindly encouraged me to submit this paper; I'm glad I followed his advice. I'm grateful to Andrei Paskevich and Evgeny Makarov for their detailed proofreadings of an early version of this paper. Finally, I wish to thank the VSTTE reviewers for their helpful comments.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
3. Bobot, F., Conchon, S., Contejean, É., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), http://alt-ergo.lri.fr/
4. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: The Why3 platform. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edn. (February 2011), http://why3.lri.fr/
5. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)

[5] Typically 32 or 64. On a practical point of view, imposing $n \leq 32$ is not an issue since the "world record", *i.e.* the largest value of n for which we have computed the solution, is $n = 26$ only [7]; we can expect all machines to be 64-bits before the limit $n = 32$ is reached, if ever.

[6] Historically, the first number of solutions announced for $n = 24$ was erroneous, due to 182 overflows on 32-bit integers — see [7] for details.

6. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
7. Technische Universität Dresden. The world record to the n-queens puzzle ($n = 26$) (2009), http://queens.inf.tu-dresden.de/
8. The Frama-C platform for static analysis of C programs (2008), http://www.frama-c.cea.fr/
9. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011), www.vscomp.org
10. Knuth, D.E.: The Art of Computer Programming, volume 4A: Combinatorial Algorithms, Part 1, 1st edn. Addison-Wesley Professional (2011)
11. Morrison, D.R.: PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM 15(4), 514–534 (1968)
12. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM 19(5), 279–285 (1976)
13. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), http://coq.inria.fr
14. Warren, H.S.: Hackers's Delight. Addison-Wesley (2003)

A Annotated Source Code

This is the annotated source code for the program in Fig. 4.

```
let rec t (a b c : set int) variant { cardinal a } =
  { 0 ≤ !k ∧ !k + cardinal a = n ∧ !s ≥ 0 ∧
    (∀ i: int. mem i a ↔
      (0≤i<n ∧ ∀ j: int. 0 ≤ j < !k →  !col[j] ≠ i)) ∧
    (∀ i: int. i≥0 → not (mem i b) ↔
      (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + j - !k)) ∧
    (∀ i: int. i≥0 → not (mem i c) ↔
      (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + !k - j)) ∧
   partial_solution !k !col }
  if not (is_empty a) then begin
    let e = ref (diff (diff a b) c) in
    let f = ref 0 in
 'L:while not (is_empty !e) do
      invariant {
        !f = !s - at !s 'L ≥ 0 ∧ !k = at !k 'L ∧
        subset !e (diff (diff a b) c) ∧
        partial_solution !k !col ∧
        sorted !sol (at !s 'L) !s ∧
        (∀ i j: int. mem i (diff (at !e 'L) !e) → mem j !e → i < j) ∧
        (∀ u: solution.
         (solution u ∧ eq_prefix !col u !k ∧ mem u[!k] (diff (at !e 'L) !e))
```

```
          ↔
          (∃ i: int. (at !s 'L) ≤ i < !s ∧ eq_sol u !sol[i])) ∧
         eq_prefix (at !col 'L) !col (at !k 'L) ∧
         eq_prefix (at !sol 'L) !sol (at !s 'L) }
      variant { cardinal !e }
      let d = min_elt !e in
      (* ghost *) col := !col[!k ← d];
      (* ghost *) incr k;
      f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
      (* ghost *) decr k;
      e := remove d !e
    done;
    !f
  end else begin
    (* ghost *) sol := !sol[!s ← !col];
    (* ghost *) incr s;
    1
  end
  { result = !s - old !s ≥ 0 ∧ !k = old !k ∧
    sorted !sol (old !s) !s ∧
    (∀ u: solution.
       ((solution u ∧ eq_prefix !col u !k) ↔
        (∃ i: int. old !s ≤ i < !s ∧ eq_sol u !sol[i]))) ∧
    eq_prefix (old !col) !col !k ∧
    eq_prefix (old !sol) !sol (old !s) }

let queens (q: int) =
  { 0 ≤ q = n ∧ !s = 0 ∧ !k = 0 }
  t (below q) empty empty
  { result = !s ∧ sorted !sol 0 !s ∧
    ∀ u: solution.
      solution u ↔ (∃ i: int. 0 ≤ i < result ∧ eq_sol u !sol[i]) }
```

Development and Evaluation of LAV: An SMT-Based Error Finding Platform[⋆]

System Description

Milena Vujošević-Janičić[1] and Viktor Kuncak[2]

[1] Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia
milena@matf.bg.ac.rs

[2] School of Computer and Communication Sciences, EPFL, Station 14, CH-1015 Lausanne, Switzerland
viktor.kuncak@epfl.ch

Abstract. We present design and evaluation of LAV, a new open-source tool for statically checking program assertions and errors. LAV integrates into the popular LLVM infrastructure for compilation and analysis. LAV uses symbolic execution to construct a first-order logic formula that models the behavior of each basic blocks. It models the relationships between basic blocks using propositional formulas. By combining these two kinds of formulas LAV generates polynomial-sized verification conditions for loop-free code. It uses underapproximating or overapproximating unrolling to handle loops. LAV can pass generated verification conditions to one of the several SMT solvers: Boolector, MathSAT, Yices, and Z3. Our experiments with small 200 benchmarks suggest that LAV is competitive with related tools, so it can be used as an effective alternative for certain verification tasks. The experience also shows that LAV provides significant help in analyzing student programs and providing feedback to students in everyday university practice.

1 Introduction

In this paper we present LAV, a tool for finding bugs (such as buffer overflows and division by zero) and for checking functional correctness conditions given by assertions.[1] We evaluated our approach primarily on programs in the C programming language (where the opportunities for errors are abundant), but it can be also used for other languages. LAV works on the LLVM low-level intermediate representation, and applies to other similar representations. LLVM[2] is a compiler framework widely used for compilation tasks, but also for verification as in tools KLEE [10], Calysto [2], and LLBMC [20]. LLVM has front-ends for C, C++, Ada and Fortran, and there are further external projects for translating a number of other languages to LLVM.

[⋆] This work was partially supported by the Serbian Ministry of Science grant 174021 and by Swiss National Science Foundation grant SCOPES IZ73Z0_127979/1.

[1] LAV stands for *L*LVM *A*utomated *V*erifier. LAV also means *lion* in Serbian.

[2] http://llvm.org/

The approach we propose combines symbolic execution [17], SAT encoding of program's control-flow, and elements of bounded model checking [6]. LAV represents program meaning using first-order logic (FOL) formulas and generates final verification conditions as FOL formulas. Each block of code is represented by a FOL formula obtained through symbolic execution (blocks have no internal branchings and no loops). Symbolic execution, however, is not performed between different blocks. Instead, relationships between blocks are modeled by propositional variables encoding transitions between blocks. LAV constructs formulas that encode block semantics once for each block. It then combines these formulas with propositional formulas encoding the transitions between blocks. The resulting compound FOL formulas describe correctness and incorrectness of individual instructions. LAV checks them using one of the several SMT solvers. If a command cannot be proved to be safe, LAV translates a potential counterexample from the solver into a program trace that exhibits this error. It also extracts the values of relevant program variables along this trace. Our experiments with 200 benchmarks suggest that LAV is competitive with related tools. The experience also shows that LAV provides significant help in analyzing student programs, providing feedback to students in everyday university practice.

2 Motivating Example

Verification tools based on symbolic execution proved to be very efficient for many verification tasks. However, they also have weaknesses that make them, in some cases, less applicable than desired. As a simple example, consider the code in Figure 1. There are four paths to be explored to check whether the program has division by zero bug in line 10. If, in line 9, *div* is assigned $a0 + a1 + 2$, then the bug occurs in the first of these four paths (assuming that `else` branches are considered first). If, on the other hand, *div* was assigned $a0 + a1 - 2$, then the bug occurs in the last of these four paths. If *div* was assigned $a0 + a1 + 3$, then there is no division by zero bug in the line 10. In summary, if there is no bug, or the bug is found in the last path, then all paths need to be explored.

If we generalize the example from Figure 1 to have n instead of two variables, and to have n instead of two `if` commands, then there are 2^n paths to consider.[3] This is the well-known problem of path explosion. Instead of considering all paths separately, our approach models the control flow in a more compact way that uses symbolic execution only within fragments (blocks) without branching. The size of a formula in this approach is polynomial in the number of blocks. Consequently, the path explosion does not occur in the verification tool itself. The exploration of many possible paths is transferred to a reasoner (i.e. theorem prover) which receives case splits only implicitly in form of disjunctions within a formula representing verification condition. Thanks to the use of learning, the reasoner typically solves such formula much more efficiently than by considering

[3] Code with this control structure is not unrealistic. Many real-world functions, such as lexical analyzers and parsers, contain a large number of `if` commands [16].

```
0:  int main()                                line 10: UNSAFE
1:  {
2:  int a0, a1, k, div = 1;                   function: main
3:  if(a0>0)                                  error: division_by_zero
4:         a0 = 1;                            3: a0 == 0, a1 == 0, div == 1
5:    else a0 = -1;                           5: a0 == -1, a1 == 0, div == 1
6:  if(a1>0)                                  6: a0 == -1, a1 == 0, div == 1
7:         a1 = 1;                            8: a0 == -1, a1 == -1,  div == 1
8:    else a1 = -1;                           10: a0 == -1, a1 == -1, div == 0
9:  div = a0+a1+2;  // div = a0+a1-2; // div = a0+a1+3;
10: k = 1/div;
11: }
```

		LAV			KLEE		
# ifs & # vars	# paths	bug in the first path	bug in the last path	no bug	bug in the first path	bug in the last path	no bug
2	4	0.07	0.07	0.07	< 1	0.05	0.05
5	32	0.18	0.19	0.18	< 1	0.55	0.55
10	1'024	0.41	0.46	0.38	< 1	45.00	45.00
11	2'048	0.42	0.54	0.43	< 1	107.00	107.00
12	4'096	0.50	0.67	0.50	< 1	268.00	268.00
20	1'048'576	0.73	1.82	0.72	< 1	TO	TO
60	2^{60}	25.00	39.00	4.18	≈ 1	TO	TO
100	2^{100}	153.00	111.00	15.00	≈ 1	TO	TO

Fig. 1. Code example (left-hand side, up) and LAV output for $div = a0 + a1 + 2$ (right-hand side, up). The table shows the number of `if`-s and variables, the number of paths, the time spent by LAV and then by KLEE if a bug occurs in: the first path, the last path, and if there was no bug. Times are given in seconds. TO means timeout.

all cases. Using this approach, we avoid the path explosion problem using modern, powerful theorem provers, such as SMT solvers. This observations motivated our approach, and shows good results in our examples. In the above example, as n increases, the time spent by our tool increases polynomially, instead of the clearly exponential growth for the symbolic execution tool KLEE [10], as shown in the table in Figure 1. The table shows that the verification of a program (in the case where there are no bugs) with large number of paths is infeasible for KLEE. The table also shows that the time needed by KLEE to find a bug (if it exists) heavily depends on the path that leads to it. Neither of these holds for our tool: verification of a correct program or finding a bug both follow a construction of a single first order formula that is passed to an SMT solver. Certain differences in the solving times (for bugs occurring in different paths) are not consequences of the modeling process, but rather of the internal operation of the solver.

3 Modelling Variables, Data Types, and Blocks

Store and Blocks. Each program function consists of blocks, while each block is a sequence of commands. The execution can enter a block only at its entry point, and exit only through the last command of the block. A *store* of a program is a map from variables to values given by variable's type. Each instruction transforms the store and may add constraints over variables. In our approach, symbolic execution is used to compute a FOL formula $Transformation(b)$ that describes how the block b transforms the store of the program.

Denote by $s(b, v)$ the value of a variable v at the entry point of the block b and by $e(b, v)$ the value of v at the exit point. After the block is symbolically executed, the formula $Transformation(b)$ is constructed based on the values of the variables at the end of the block. In the general case it is given by $\bigwedge_{v \in V}(e(b, v) = e_v) \bigwedge AdditionalConstraints(b)$, where V is the set of variables and where e_v is the value of v at the end of the block expressed in terms of initial values $s(b, v)$. The formulas $AdditionalConstraints(b)$ are introduced for modeling certain operations (as described in the rest of this section). Another formula $Transformation(b, i)$ is defined analogously, but considering only the first i instructions of the block b.

Buffers, Structures and Unions. Buffers are sequences of memory allocated statically or dynamically and accessible by a pointer and an offset. While these pointers are treated in our tool as any other simple variables, they are also associated with sizes of corresponding buffers. To deal with buffer sizes, we introduce two functions: $left(p)$ and $right(p)$ for numbers of bytes reserved for the pointer p on its left and its right hand side. For example, the command `*(p+i)` introduces a buffer overflow iff $left(p) \leq i \cdot sizeof(int) < right(p)$ is false. The argument of the functions $left$ and $right$ can be a pointer or a sum of a pointer and an offset (which is of the integer type), in accordance with properties of pointers in C. Note that it always holds $left(p + n) = left(p) - n$ and $right(p+n) = right(p) - n$. These equalities can be considered as axioms about the functions $left$ and $right$, but, instead of introducing a universally quantified formula into the generated formula, we add only its relevant instances to the set of additional constraints attached to the block. More complex types, such as structures and unions, are also treated as sequences of individual bytes.

Memory Contents. For simplicity and precision, LAV uses a flat memory model: it treats the entire memory as an array *mem* of memory locations, that may get updated during the symbolic execution, just as any other variable. For modeling commands that access the memory via pointers we use the theory of arrays. The theory of arrays provides functions for storing a value at a certain index in the array (*store*) and for reading a value at a certain index in the array (*select*). Also, if there is a reference operator on a local variable within a function, then this variable is not tracked through its slot in the store, as other variables, but is tracked through the memory content. A run-time library guarantees that during the program execution all active variables and dynamic objects are assigned non-overlapping memory locations. Instead of adding conditions of the form $p \neq q$ for each pair (p, q) of addresses of variables or dynamic objects, a more efficient approach is used: each address p is assigned (within correctness conditions) a unique *fixed* number (or *magic number* [19]).

Global Variables. Global variables are accessible in all functions (and, hence, in all blocks), but instead of representing them individually within all functions, they are modeled by the variable modeling memory. The reasoning involving the theory of arrays can be expensive, but if there are not many updates of global variables, this model can still be more suitable. If there are many updates of global variables, then, in some cases, global variables are tracked through their slots in the store, just as local variables.

Function Calls. Function calls are modeled according to the available information about the function. If a *contract* (i.e., a *summary*) of the function is available, then the current store is updated and additional constraints are added according to this contract. If a contract of the function is not available, while the definition of function is, then an interprocedural analysis is required (performed as described in the next section). If neither a contract nor the definition of the function are available, then the memory contents (i.e. the current array *mem*) is set to a new (fresh) variable as an effect of the function call.

4 Modelling Control Flow

Intraprocedural Loop-free Control Flow. Whereas single blocks are represented by FOL formulas constructed using symbolic execution, LAV encodes relationships *between* blocks by propositional variables and SAT formulas.

Assume, for a moment, that the program has no loops in the control-flow graph. A path in this graph is then determined by the sequence of nodes (representing blocks) and edges (representing transitions from one block to another). For each block and for each transition we introduce a propositional variable that denotes whether the corresponding node or transition is in the path. Valid paths through the graph are encoded by entry conditions (represented by $EntryCond(b)$) and exit conditions ($ExitCond(b)$). Entry conditions are conditions that must hold at the entry point of the block b — b is in the path iff there is a transition to b from some of its predecessors; if the block b is reached from the block *pred*, then the initial values of the variables within the block b are equal to the values of the variables at the exit point of the block *pred*. Exit conditions are conditions that must hold at the exit point of the block b — each block must lead somewhere (either to some other block or to the exit of the function). If the block b was active and if the exit condition c_i of the block b was met, then the control is passed to the successor $succ_i$. The final formula *Description(b)* describing the block b is defined as $EntryCond(b) \ \wedge \ Transformation(b) \ \wedge \ ExitCond(b)$.

Loops. Loops are eliminated by unrolling. This way, the control flow graph of the function has no cycles and the above modeling mechanism can be applied. Our system supports two techniques for dealing with loops: underapproximation of loops and overapproximation of loops. In the former case, loops are unrolled a fixed number of times n, as in bounded model checking. If the unrolled code verifies successfully, it means that the original code has no bugs for n or less passes through the loop. In the latter case, the unrolled code simulates first n and last m entries to the loop, where n and m are configurable parameters. After the first n unrollings, we insert a block of code which simulates execution of an arbitrary loop iteration by resetting the values of each loop target (i.e. the value of each variable that is updated by any statement in any block in the loop), similarly as, for instance, in [4]. This resetting of the values may cause loss of precision and therefore may introduce false alarms. To overcome this problem, it is necessary to have loop invariants annotated in the program (or to use techniques that infer them automatically in some cases). If the overapproximated code is verified, then the original code has no bugs too.

Interprocedural Control Flow. Starting with block descriptions as building blocks, and given that there is a unique entry and a unique exit point for each function,[4] the description of a function is constructed as a conjunction of descriptions of the function blocks. Recursive function calls can be unrolled in a similar way as loops.

5 Correctness Conditions

To check whether a command leads to an error LAV builds two formulas, of the form $C \Rightarrow (\neg)safe(c)$. Here C is a formula describing a context: in the empty context (i.e., if the command is considered on its own), C equals $\top$, in the block context, C equals $Transformation(b, i)$ (if c is i-the instruction in b), and in the function context, C equals $\bigwedge Description(b')$ for all function's blocks b' that precede b. $(\neg)safe(c)$ is a formula describing (in)correctness condition of a command — it can be given by a bug definition (division by zero, buffer overflow, dereferencing null pointers) or it can be given by an annotation in the form of C logical expressions within assert commands.

If $safe(c)$ holds (under assumption C), then the command is *safe* and if $\neg safe(c)$ holds, the command c is *flawed*. If both $safe(c)$ and $\neg safe(c)$ hold for some context C (i.e. if C is inconsistent), then the command c is *unreachable*.[5] If neither $safe(c)$ nor $\neg safe(c)$ hold in a general case, then the command c is considered *unsafe*. The difference between a flawed and an unsafe command is that the flawed command always leads to an error in the program (if it is reachable), while unsafe command leads to an error only in some cases, depending on the context of the command, i.e., to the path condition leading to the command. Each command is first checked within the empty context. If it gets the unsafe status, the command is then checked within the block context. If it keeps the unsafe status within the block context, then it is checked within the function context. If a command is detected to be safe or flawed at some stage, then this status for the command is final and wider contexts are not considered. For each function call, correctness conditions for all unsafe commands from the called function are checked in the calling context.

Checking the status of the command c in the context C can always be done within one or two prover calls. After the context is added into the solver, the safety property of the command is first checked. If the solver proves it, then this means that it is either safe or unreachable. In both cases, it is not flawed or unsafe so there is no need for any further checks. If the solver cannot prove it,

[4] This can be ensured, as in the LLVM code.

[5] Note that, even if it was proved that the command c is safe, unsafe or flawed in some context, it still does not mean that it is reachable in some wider context.

then the negation of the safety property is checked and according to the answer of the prover it can be concluded if it is a flawed or an unsafe command. If one does not want to distinguish between flawed and unsafe commands (by selecting this tool's option, in case when trade-off of solving time and precision is needed), then this second call is omitted.

When proving different (in)correctness conditions in one function context, formulas corresponding to unnecessary but already considered blocks can be kept in the context (thanks to deductive monotonicity). This enables incremental approach, suitable for SMT solvers that can typically take advantage of the results learned from the previous proof attempts [5].

6 Transforming a Code Model into an SMT Goal

The quantifier-free formula that models program code typically involves arithmetic, logical, and relational operators, but also functions such as $left$ and $right$. We model integers by arbitrary-precision numbers (using linear arithmetic) or, if so selected by a command-line argument, by finite-precision numbers in bit-vector arithmetic. The functions $left$ and $right$ are considered to be *uninterpreted functions*, with their specific properties added to the correctness conditions. These functions are dealt by: (i) the theory of uninterpreted functions or by (ii) Ackermannizing the goal [1]. Each of these options can lead to more efficient reasoning in some cases [8]. Memory contents are represented by the theory of arrays, or can be just ignored (leading to a less precise reasoning) because of a high computational cost. Overall, the models of code typically require: bit-vector arithmetic (or linear arithmetic), theory of uninterpreted functions (or, alternatively, Ackermannization), and optionally theory or arrays. There are several SMT solvers that provide support for such combinations of theories.

7 Implementation

The approach described in previous sections is implemented in C++ as a tool LAV. The tool is publicly available and open-source.[6] The tool is built on top of LLVM that serves as a front-end to input programs. LLVM is developed primarily for the programming language C, but can be used for other languages as well. Thanks to this universality, LAV successfully handled several non trivial examples in Fortran. (Dealing with object oriented languages requires certain additions to our tool, which are planned for our future work.)

The LLVM programs are processed and the formulas representing correctness conditions are generated following the approach described in the sections 3 and 4. As the default parameters, loop unrolling simulates the first two and the last one passes through the loop, but this can be changed by the user. Correctness conditions, built as described in Section 5, can be translated to a number of theories and their combinations, as described in Section 6. Currently, there is

[6] `http://argo.matf.bg.ac.rs/?content=lav`

support for export to linear arithmetic, bit-vector arithmetic, the theory of uninterpreted functions (and Ackermannization, as its alternative), and the theory of arrays. Recursive function calls and support for floating point number arithmetic are not implemented yet. Automated inference of loop invariants is not part of LAV. There is currently no general notation for function contracts, but contracts of certain memory-safety-critical functions such as `malloc`, `calloc`, `realloc`, `free`, `strcpy` are encoded directly in C++, within LAV implementation. In addition, statements or assumptions (concerning loops or function calls) can be given in the form of C logical expression within *assume* function call.

The generated formulas are passed to SMT solvers, by using function calls from their APIs. Currently, supported solvers are Boolector [7] (for the theories BVA and ARRAYS), Yices [15] and MathSAT [9] (LA, BVA and EUF) and Z3 [14] (LA, BVA, EUF, ARRAYS). For unsafe and flawed commands, a counterexample which includes program trace and values of program variables along this trace is extracted from the model generated by a solver (if a corresponding option is used).

8 Evaluation and Comparison to Related Tools

Related tools. CBMC [12] and ESBMC [13] are bounded model checkers for ANSI C programs. As a front-end, ESBMC uses CBMC, which, in turn, uses `goto-cc`, a compiler from C and C++ into GOTO-programs. On the other hand, LAV uses LLVM, which is a multi language platform. CBMC and ESBMC unwind program loops, while LAV supports both underapproximation and overapproximation of loops. CBMC translates correctness conditions to propositional logic and instances of SAT. Like LAV, ESBMC converts verification conditions using different background theories and passes them directly to an SMT solver.

KLEE, Calysto, S2E, and LLBMC use the LLVM compiler infrastructure as a front-end to input programs. KLEE [10] is a symbolic execution tool, which employs a variety of constraint solving optimizations, represents program states compactly and uses search heuristics to improve code coverage. KLEE is used within other verification tools, such as the S2E platform [11] for developing analyzers. S2E introduces selective symbolic execution, relaxed execution consistency models, and supports analysis of binaries. Calysto [2] is a static checker for NULL pointer dereferencing and user-provided assertions. Calysto preserves the structure of the analyzed program in the phase of symbolic execution and uses it as an automatic abstraction/refinement framework for filtering verification conditions. It handles loops and pointers in an unsound manner; for example, loops are unrolled only once. The above tools are closely integrated with their theorem provers and with theories that these provers use. This is in contrast to LAV, which can chose amongst different SMT solvers and theories. LLBMC [20] is a tool for low-level bounded model checking of C programs, which was developed in parallel with our work. It focuses on covering memory consistency constraints; it models control flow of a program in a similar way as LAV and uses Boolector (the theory of bit-vectors and arrays) as the back-end solver.

Table 1. Frontends, supported theories and solvers for considered tools

Tool	LAV	CBMC	ESBMC	KLEE	LLBMC	CALYSTO	PEX
Frontend	LLVM	goto-cc	goto-cc	LLVM	LLVM	LLVM	.NET
Theories	-	PL	-	-	-	-	-
	LA	-	LA	-	-	-	LA
	BV	-	BV	BV	BV	BV	BV
	EUF	-	EUF	-	-	-	EUF
	ARRAYS	-	ARRAYS	ARRAYS	ARRAYS	-	ARRAYS
Solvers	MathSAT	MiniSAT2	CVC	STP	-	Spear	-
	Boolector	-	Boolector	-	Boolector	-	-
	Z3	-	Z3	-	-	-	Z3
	Yices	-	-	-	-	-	-

Table 1 summarizes the front ends, supported theories, and solvers used by considered tools.

Experimental Comparison. We describe experimental comparison of LAV with KLEE, CBMC and ESBMC. At the time of writing, LLBMC and Calysto are not publicly available, so we were not able to include them in this evaluation. We also did not include the symbolic execution tool PEX [21], because it deals with C# and not with C.

The experimental comparison of LAV with the related tools was based on the NECLA static analysis benchmarks [19]. These benchmarks contain C programs that demonstrate common programming situations that arise in practice such as interprocedural data-flow, aliasing, array allocation modes, array size propagation, string library usage and so on. The ability of different techniques to prove them (in)correct is an indication of their areas of strengths and weaknesses. All ANSI C programs from the NECLA static analysis benchmarks are included in our evaluation except those that contain recursive function calls, string library usage and which depend on floating point number calculations (44 out of 57 benchmarks are included).

All the tools checked the benchmarks for pointer errors, buffer overflows, division by zero, and user-defined assertions. The tools terminated (with an appropriate report) when a first flawed command was found or when the code was verified. The experiments were performed with default parameters for each tool. We consider the results obtained with default parameters the most indicative since the user does not have to examine the code in order to determine unwinding and other parameters. If some tool, for its default parameters, produced an irregular output (such as an error message, a false alarm or time out), then it was invoked again with a loop unwinding parameter — with the upper bound of the loop, if it exists. If that call produced an irregular output or the upper bound of the loop does not exist, then a small loop bound was used. LAV and ESBMC were used with a solver for the theory of bit-vectors and arrays (because for KLEE this is the only option).

Experiments were performed on a system running Ubuntu, with Intel processor on 1.6GHz and with 1GB of RAM memory. The results are given in Table 2.

The table contains a name of the benchmark (`bnc`), the number of code lines (`#L`), the number of loop unwindings (`#UNW`), whether or not the program contains some flawed commands (`F/V`), the name of the tool and the name of the solver used (for some tools, in some cases, the verification did not require invoking a solver). Abbreviations used are: `B` – Boolector, `NA` – not applicable, `FA` – false alarm, `UB` – missed (not discovered) bug, `U` – unreachable bug, `*` – SAT/SMT solver was not called, `ERR` – error, `TO` — time out, and `Z3` – solver Z3 was called instead of Boolector. The summary of the results is given in Table 3.

Table 2. Experimental results

bnc.	`#L`	`#UNW`	`F/V`	LAV		CBMC	ESBMC		KLEE
				B	Z3		B	Z3	
ex1	21	def	V	FA	FA	**5.02***	**5.02***	**5.02***	**0.19**
		513	V	TO	TO	**5.02***	**5.02***	**5.02***	NA
		3	V	0.90	0.35	**0.14***	**0.14***	**0.14***	NA
ex2	40	def	V	0.63	**0.54**	TO	TO	TO	ERR
		1024	V	TO	TO	ERR	Z3	**67.48**	NA
		3	V	1.03	0.47	ERR	Z3	**0.27**	NA
ex3	24	def	F	**0.04**	0.06	0.08	0.09	0.09	**0.04**
ex4	16	def	F	0.13	0.24	0.14	0.15	0.18	**0.02**
ex5	18	-	V	**0.02**	**0.02**	0.06*	0.06*	0.06*	**0.02**
ex6	21	-	V	0.07	0.11	0.07	Z3	0.07	**0.03**
ex7	28	def	V	**0.22**	**0.22**	TO	TO	TO	ERR
		3	V	0.21	**0.15**	ERR	Z3-FA	FA	NA
ex8	20	def	F	**0.13**	0.15	TO	TO	TO	ERR
		3	F	**0.14**	**0.14**	FA	ERR	ERR	NA
ex9	43	def	V	1.34	**0.85**	TO	TO	TO	ERR
		1024	V	TO	TO	ERR	Z3-TO	TO	NA
		3	V	2.93	**0.62**	ERR	Z3-FA	FA	NA
ex10	72	def	F	1.32	0.59	TO	TO	TO	**0.03**
		17	F	TO	10.47	**0.31**	UB	UB	NA
		3	F	4.02	1.14	**0.13**	UB	UB	NA
ex11	25	def	V	FA	FA	TO	TO	TO	TO
		3	V	**0.05**	0.08	0.06*	0.06*	0.06*	NA
ex12	24	def	F	0.12	0.16	0.12	0.10	0.10	**0.03**
ex13	10	-	F	**0.03**	0.44	0.07	0.06	0.13	TO
ex14	16	def	V	0.10	0.13	0.08*	0.08*	0.08*	**0.03**
ex15	35	-	V	0.56	0.34	FA	Z3	0.09	**0.03**
ex16	35	def	U	**0.09F**	0.10F	TO	TO	TO	TO
		2	U	**0.08F**	0.09F	**0.08*V**	**0.08*V**	**0.08*V**	NA
ex17	45	def	V	1.56	0.68	0.34*	0.24*	0.24*	**0.02**
ex18	30	def	V	FA	FA	TO	TO	TO	ERR
		100	V	TO	TO	ERR	ERR	ERR	NA
		10	V	**1.30**	3.0	ERR	ERR	ERR	NA
ex19	29	def	V	FA	FA	TO	TO	TO	TO
		3	V	0.14	**0.08**	0.10	**0.08**	0.09	NA
ex20	33	def	F	FA	FA	TO	TO	TO	**0.12**
		1024	F	455	TO	40.98	**40.0**	206	NA
		3	F	0.21	0.32	0.25	**0.09**	0.11	NA
ex21	26	def	V	0.45	0.36	FA	Z3	1.68	**0.02**
ex22	39	def	V	12.22	4.1	0.64	Z3	0.81	**0.06**
ex23	26	def	V	FA	FA	16.49	**0.16**	0.18	0.69
		36	V	25.14	6.46	16.49	**0.16**	0.18	NA
ex25	27	def	V	**0.26**	0.27	TO	TO	TO	TO
		3	V	0.21	0.20	0.10*	**0.08***	**0.08***	NA
ex26	30	def	F	1.88	**0.62**	6.42	Z3	4.79	UB
ex27	40	def	F	25.34	5.28	3.40	Z3	3.24	**0.09**
ex30	44	def	F	**0.15**	0.24	TO	TO	TO	ERR

Table 2. (*Continued*)

bnc.	#L	#UNW	F / V	LAV B	LAV Z3	CBMC	ESBMC B	ESBMC Z3	KLEE
		100	F	TO	TO	ERR	Z3-UB	UB	NA
ex31	14	def	V	FA	FA	TO	0.08*	0.08*	**0.02**
		7	V	1.38	5.62	0.57	**0.08***	**0.08***	NA
ex32	27	def	V	0.78	0.5	2.30*	Z3	4.11	**0.18**
ex34	25	-	V	**0.08**	0.24	0.10	0.12	0.14	0.16
ex37	30	-	F	**0.16**	0.20	FA	Z3-UB	UB	ERR
ex39	27	def	F	**0.06**	0.08	TO	TO	TO	ERR
		3	F	**0.07**	**0.07**	UB	0.09	0.09	NA
ex40	20	def	V	0.09	0.12	TO	TO	TO	**0.02**
		3	V	**0.08**	0.10	0.12	**0.08**	**0.08**	NA
ex41	23	def	F	0.25	**0.25**	TO	TO	TO	17
		3	F	0.49	0.44	0.25	**0.07**	0.10	NA
ex43	113	def	F	28.56	17.91	FA	Z3	25.15	**0.06**
ex46	38	def	F	6.57	**5.75**	TO	TO	TO	ERR
		2	F	**37.43**	TO	FA	Z3-FA	FA	NA
ex47	35	def	F	3.71	**2.32**	TO	TO	TO	ERR
		2	F	4.40	**1.38**	FA	Z3-FA	FA	NA
ex49	16	def	F	0.18	**0.11**	TO	TO	TO	TO
		3	V	**0.06**	0.08	0.08	0.07	0.08	NA
inf1	36	-	F	**0.12**	0.22	0.18	UB	0.15	0.13
inf2	63	-	F	4.84	**1.25**	FA	Z3-FA	FA	UB
inf4	62	-	F	0.23	0.38	**0.11**	0.12	0.19	0.40
inf5	62	-	F	0.09	0.15	0.11	0.12	0.15	**0.06**
inf6	43	-	V	0.29	0.12	0.41	Z3	0.40	**0.06**
inf8	44	-	V	0.16	0.19	0.11	FA	0.12	**0.06**

Analysis of Results. False alarms and bugs undiscovered by CBMC can be explained by the way it models memory and control-flow of the programs. For example, CBMC assumes that each dynamic memory allocation will succeed, although this is not valid assumption. Also, CBMC does not precisely model the memory assigned to global arrays and pointers to pointers so this explains some false alarms. Concerning control-flow, imprecisions may arise after loop unwinding. If CBMC cannot prove that the unwinding is performed for the upper loop bound, it dismisses all current information about memory allocations for arrays, no matter if these allocations were static or dynamic. Since CBMC reports only flawed commands (and not commands that cannot prove to be safe), this may lead to undiscovered bugs. CBMC does not check/report unreachable bugs. Therefore, it can miss a bug if it is unreachable for all CBMC-feasible unwinding parameters.[7]

ESBMC inherits program modeling of CBMC, but also introduces some improvements. ESBMC models memory more precisely and it exhibits less false alarms than CBMC. It models that dynamically allocation may not succeed, but it still may miss some NULL-dereferencing bugs. Concerning solvers, it seems that ESBMC does not have support for Ackermannization and it calls the Z3

[7] For instance, in example 39.c, CBMC encounters time out for its default parameters, and for a small number of loop unwindings it does not discover the bug. The bug in this example, which is reachable as a consequence of a possible integer overflow, is discovered by LAV even for a small number of loop unwindings because LAV finds that the command itself is flawed so its reachability is not further checked.

Table 3. Summary

Tool	LAV	CBMC	ESBMC	KLEE
Best times with default params.	45%	2%	0	**47%**
Best times with upp.bound	0%	22%	**56%**	NA
Best times with unw.bound	**66%**	17%	44%	NA
Timeouts	**11%**	26%	26%	13%
False alarms	9%	11%	8%	**0%**
Errors	**0%**	11%	4%	23%
Undiscovered bugs	**0%**	1%	7%	4%

solver instead of Boolector whenever the theory of uninterpreted functions is involved (Boolector does not support the theory of uninterpreted functions). Also, it is likely that there are some errors in solver interfaces since ESBMC can give different results when different solvers for the same theory are used for the same problem. ESBMC exhibits the largest number of timeouts and the largest number of undiscovered bugs. ESBMC with its default parameters was not best for any benchmark time, but it has the largest number of best times when the upper loop bound was specified.

The usage of KLEE is somewhat different than the usage of LAV, CBMC and ESBMC. Unlike these tools, for KLEE it is necessary to annotate programs with claims which variables should be treated as symbolic. It is not possible to have dynamic memory allocation with a size represented by a symbolic value, so KLEE reports error messages for some benchmarks. Also, it is not possible to simulate nondeterministic choice as a loop entry parameter. KLEE does not terminate when it finds a first error (as other tools do) and there is no option to do so. However, this behavior does not affect the best times reported in the table. As a symbolic execution tool, the number of loop unwindings cannot be specified to KLEE, but the number of states considered can. Since this is not comparable to number of loop unwindings, we compare KLEE to other tools only with its default parameters. KLEE had six time outs, ten errors, two undiscovered bugs and no false alarms. It has the largest number of best times. KLEE was the most efficient on examples where there is only one possible path through the program, because it efficiently finds it and the symbolic execution in these cases take almost the same as a real execution. Other tools, because of the different nature of modeling, do not take the advantage of having just one path through the code. However, in practical applications, this is rarely the case.

LAV has no timeouts for its default parameters. This comes with a price of the biggest number of false alarms for default parameters. These false alarms are due to the policy of LAV that reports all commands that are potentially unsafe (that could not be proved to be safe). So, all the false alarms come with a message that the command is potentially unsafe, and never with a message that the command is flawed, which makes the difference to tools that have no ranking of potential bugs. If we change this policy, and if LAV reports only commands which are proved to be flawed, then LAV would not have so many false alarms but would have undiscovered bugs. In more than half cases when LAV reported false alarm, the other tools had timeouts, therefore, neither tool exhibited desired behavior. Concerning timeouts, most of the timeouts that LAV exhibited were due to the high loop unwinding parameter. In most cases, the default parameters already gave good results so there was no need for the unwinding with the upper loop bound. LAV has no error messages, undiscovered bugs and has no false alarms for a fixed number of unwindings. If we compare Boolector and Z3, we can see that efficiency of LAV with Boolector is very similar to the efficiency obtained with Z3, except that there are two cases when Z3 encountered timeout when Boolector did not and only one case when Boolector encountered timeout and Z3 did not.

We believe that these results present a useful experimental comparison of existing tools. They also suggest that LAV is an interesting point in the design space of verification and bug finding tools.

9 Application in Education

Software verification tools have different areas of application. One typical area of application are safety-critical computer programs. On the other hand, verification tools can be very beneficial in checking computer programs that are far from being safety-critical but are massive in number and have other nature of importance. In this section we consider one such application: computer programs developed by students within programming courses in high schools and universities. A tool that could help students and teachers to notice errors in programs would have multiple benefits. For students, such tool would be helpful when there is no teacher to check the solution (which is, most of the time, the case). For teachers, such tool would be helpful in marking exams, at least for pointing to standard errors. For both, such tool would be illustrative and would demonstrate power of verification tools, to which students should get accustomed and ready to adopt in their professional work.

With this motivation in mind, we performed another set of experiments with our tool—we analyzed programs written by students that took an introductory C programming course at the University of Belgrade. Our corpus consists of 157 programs which were written by the students at test exams along the course.[8] We divided the corpus into three groups. The first group consists of solutions of problems that involve numerical calculations and manipulation of command line arguments. The second group consists of solutions of problems that involve manipulation with arrays and matrices. The third group consists of solutions of problems that involve manipulation of strings and data structures. LAV was set to use its default parameters and the time that LAV spent in analyzing the programs was typically negligible. Some of the programs from the corpus did not meet the given specification, but we considered only bugs and not functional correctness. LAV discovered 423 genuine bugs in 121 programs and had 32 false alarms in 8 programs.

[8] The corpus can be found at the LAV web page.

Table 4. Application in education: the table contains the number of solutions considered for the given problem, the average number of lines per solution, the average number of reported bugs per solution, and the average number of false alarms per solution

Problem	# Solutions	Avg. Lines	Avg. Reported Bugs	Avg. False Alarms
calculations	60	30	0.82	0.05
arrays and matrices	71	46	4.20	0
strings and structures	26	60	2.92	1.11
Summary	157	42	2.69	0.20

The results of our experiments are summarized in Table 4. In the first two groups, the largest number of bugs were possible buffer overflows (225 bugs were discovered in 81 programs), while the next most frequent bug was division by zero (22 bugs in 22 programs). In the third group, the largest number of bugs were possible null pointer dereferencing (46 bugs in 15 programs), while the next most frequent bug was buffer overflow (30 bugs in 15 programs).

The vast majority of bugs that students produced follow wrong expectations — for instance, expectations that input parameters of their programs will meet certain constraints and that memory allocation will always succeed. In many cases, omission of a necessary check (e.g. whether an input parameter meets the constraints) produces several bugs in the rest of the program. This explains the large number of bugs in the corpus — adding only one check in a program would typically eliminate several bugs. Apart from these sources of bugs, there were just a few bugs with other origin (such as uninitialized variables or accessing

```
1:  #include<stdio.h>
2:  #include<stdlib.h>
3:  int power(int n)
4:  {
5:  int i, pow;
6:  for(i=0, pow=1; i<n; i++, pow*=10);
7:  return pow;
8:  }
9:
10: int get_digit(int n, int d)
11: {
12: return (n/power(d))%10;
13: }
14:
15: int main(int argc, char** argv)
16: {
17: int n, d;
18: n = atoi(argv[1]);
19: d = atoi(argv[2]);
20: printf("%d\n", get_digit(n, d));
21: }
```

```
line 12: UNSAFE
line 18: UNSAFE
line 19: UNSAFE
line 20: 12: UNSAFE

function: get_digit
error: division_by_zero
line 12: d == 1073741824,

function: main
error: buffer_overflow
line 18: argc == 1, argv == 1

function: main
error: buffer_overflow
line 19: argc == 2, argv == 1

function: main
error: division_by_zero
line 20: 12: argc == 512,
             argv == 1,
             d == 1073741824, n == 0
```

Fig. 2. A simplified version of a program from the corpus (shown on the left) and the LAV output (shown on the right). The shown output for this program is generated by invoking LAV with default parameters, so the loop in line 6 is over-approximated. The output states that lines 12, 18 and 19 are unsafe in general, and that the line 12 is unsafe when the function `get_digit` is called from the line 20. LAV also shows the nature of a possible error and the values of relevant variables.

memory that was not allocated). Concerning false alarms, all false alarms were consequences of overapproximations of loops or of the absence of support within LAV for some functions from the standard library. It was beyond the scope of this experiment to manually annotate all existing bugs so we cannot report on the number of bugs that LAV missed to report.

A simplified example of a program from the corpus is given in Figure 2. This example illustrates several typical students' bugs: two possible buffer overflows (lines 18 and 19) and one division by zero (line 12). Models generated by LAV (given at the right-hand side of the figure) can help in correcting these bugs. Although these are only preliminary experiments, the experience suggest that LAV can be useful in everyday practice of an introductory programming course.

10 Conclusions and Further Work

We presented a new software verification approach and a corresponding tool, LAV, for bug finding and for checking correctness conditions. LAV uses the compiler intermediate language, LLVM, code. Therefore, LAV need not deal with the specifics of C, and can be used for analysis of programs in several programming languages. In addition, the approach can be used with any similar low-level code representation. The approach combines symbolic execution, SAT encoding of program's behavior and bounded model checking. Individual blocks of the code are modeled by first-order logic formulas constructed by symbolic execution, and relationships between blocks are modeled by propositional formulas. Formulas that describe blocks' behavior are combined with correctness conditions for individual commands to produce correctness conditions of the program to be verified. These conditions are passed on to an SMT solver covering a suitable combination of theories. The proposed approach is implemented as an open-source tool LAV. Currently, several SMT solvers (Boolector, MathSAT, Yices, and Z3) are supported. Our experiments suggest that the presented approach is competitive with related tools. We believe that our approach can be a useful component of tools that combine multiple analysis and model checking approaches (as suggested by recent tools [3]).

In the future we plan to further improve the modeling power and efficiency of the tool. We plan to modify the implementation to use multi-core processor design. We also plan to improve our interprocedural analysis and the robustness of the tool. We are interested in further experiments with other benchmark suites (such as, for example, [22] and [18]). We plan to make a number of extensions, such as an improved user interface using a web client, more descriptive bug explanations, and automated test-case generation. We expect that these features will make LAV even more applicable in education and practice.

References

1. Ackermann, W.: Solvable cases of the decision problem. North-Holland (1954)
2. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: ICSE 2008, pp. 211–220. ACM (2008)

3. Balakrishnan, G., Ganai, M.K., Gupta, A., Ivancic, F., Kahlon, V., Li, W., Maeda, N., Papakonstantinou, N., Sankaranarayanan, S., Sinha, N., Wang, C.: Scalable and precise program analysis at nec. In: FMCAD (2010)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. ACM Sigsoft Software Engineering Notes 31, 82–87 (2006)
5. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
6. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58 (2003)
7. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
8. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Santuari, A., Sebastiani, R.: To Ackermannize or Not to Ackermannize? In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 557–571. Springer, Heidelberg (2006)
9. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
10. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
11. Chipounov, V., Kuznetsov, V., Candea, G.: S2e: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput. Archit. News 39 (2011)
12. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSL-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
13. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ansi-c software. In: ASE, pp. 137–148 (2009)
14. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
15. Dutertre, B., de Moura, L.: The Yices SMT solver. Tool paper at (August 2006), `http://-yices.csl.sri.com/tool-paper.pdf`
16. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Proc. ACM SIGPLAN POPL (January 2001)
17. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
18. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: Proceedings of ASE 2007. ACM (2007)
19. Sankaranarrayanan, S.: Necla static analysis benchmarks (2009), `http://www.nec-labs.com/research/system`
20. Sinz, C., Falke, S., Merz, F.: The low-level bounded model checker llbmc: A precise memory model for llbmc. In: SSV (2010)
21. Tillmann, N., Halleux, J.: Pex White Box Test Generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
22. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Softw. Eng. Notes 29 (2004)

A Lightweight Technique for Distributed and Incremental Program Verification

Martin Brain[1,2] and Florian Schanda[2]

[1] Department of Computer Science, University of Bath, Bath, BA2 7AY, UK
`mjb@cs.bath.ac.uk`
[2] Altran Praxis Limited, 20 Manvers Street, Bath, BA1 1PX, UK
{`martin.brain,florian.schanda`}`@altran-praxis.com`

Abstract. Applying automated verification to industrial code bases creates a significant computational task even when the individual conditions to be checked are trivial. This affects the wall clock time taken to verify the program and has knock-on effects on how the tools are used and on project management. In this paper a simple and lightweight technique for adding incremental and distributed capabilities to a program verification system is given. Experiments with an implementation of the technique for the SPARK tool set show that it can yield an average 29 fold speed increase in incremental use and near optimal speedup in distributed use. Critically, this gives a qualitative change in how automated verification is used in a large commercial project.

1 Introduction

When using program verification tools in an industrial context, it is not uncommon to have code bases that are significantly larger than those reasonable to create for academic study. For example, the commercial application used in this paper's experiments consists of 890,000 physical lines / 260,000 logical lines of SPARK code. Working at this scale creates additional problems with the time taken to verify the software. The code base in question generates over 126,000 verification conditions. Even if it were possible to resolve each of these in 0.1 seconds, verification would still require over 3 CPU hours.

In a typical development model the verification process will be run many times while the code is under development. Thus the time taken to verify makes a significant difference to how the tools are used and how the project is managed. To illustrate this issue it is useful to think of verification time in terms of the qualitative time bands given in Table 1.

The two interactive time bands ("Typing" and "Wait") can be viewed as part of the process of writing the code and are run without a shift in the developer's attention. Adopting such tools is similar to migrating to a new IDE and development suite. The next two time bands involve a shift in attention and are thus typically run once a modification has been implemented. This is similar to a change in development process, such as requiring all regression tests pass before code is committed. The asynchronous time bands ("Daily" and "Nightly")

Table 1. Qualitative time bands

Category	Band	Description	Limit
Interactive	Typing	Can be run as part of an IDE. For example basic syntax checking, some type checking and template based bug detection [21].	1s
	Wait	Can be run while the developer waits. Includes most slower lightweight static analysis tools.	30-60s
Synchronous	Coffee	Can be run while the developer switches to a short term alternative task. Compilation of reasonably sized application and build time checks often fall into this category.	10m
	Lunch	Can be run while the developer switches to an alternative task. This include running reasonably sized unit and regression test suites.	60m
Asynchronous	Daily	Can only be run once or twice per working day, requiring developers to reorganise the day-to-day development process around the tool's usage.	4h
	Nightly	Only practical to run overnight. If verification is required before a modification can be accepted then the minimum time to make a code changes is two days.	12h
Phase	Weekly	Can be run each weekend. No longer reasonable to tie verification to low level incremental changes.	2d
	Phase	Requires a separate development phase and changes in how the project is scheduled and managed.	∞

require major changes to developers' day-to-day practices and thus require a different approach to project management. The results of verification will also typically be returned after the detail of the modification has passed out of the developer's short term memory. In the final two bands reviewing the results becomes a task in itself and is effectively separated from the original modifications. If decreasing the wall clock time taken results in dropping one or more bands, it can give a significant decrease in project costs.

Most research on making verification tools faster focuses on improving the proof tools to reduce the latency of proving individual formulae. However, for industrial sized code bases there is a need to increase the throughput of proven formulae to reduce the wall clock time of the verification process. This paper presents two contributions towards this high level problem:

– First, Section 3 describes a lightweight technique for incremental and distributed proof. This makes use of memcached [16], a tool originally designed for high performance web applications. A prototype implementation of this approach for the SPARK tool chain required less than 250 lines of code.
– Next, Section 4 presents experimental results on an industrial application showing how the implementation can be used to achieve an average 29 fold speedup during incremental use and near optimal speedup during distributed use. Critically, this changes the qualitative time band for a large scale industrial project from "Nightly" to "Coffee".

2 The SPARK Language and Tools

It is useful to appreciate the nature and scale of the reasoning problems that are generated during verification of industrial code bases. This shows why the latency of proof tools on individual conditions is less of an issue than overall verification throughput. To this end this section presents an overview of the SPARK language and SPARK tool set and statistics on its usage on a large industrial application.

2.1 The SPARK Language

SPARK is a subset of Ada with an additional contract language. Variants exist for Ada 83, Ada 95 and Ada 2005. The key aim behind the creation of SPARK was to create a language that increases the likelihood of a program behaving as expected [2]. Restrictions were made to remove language features that were ambiguous, platform or implementation specific or had unclear semantics and to make formal reasoning (manual, semi-automated and fully automatic) about programs achievable. These restrictions are common to many high integrity language subsets and include limitations on access types (similar to pointers in C), goto statements, recursion, overloading, dynamic dispatch and dynamic memory. However unlike many languages for high integrity software development, SPARK has support for concurrent programming by including a subset of the Ravenscar profile of the Ada standard called RavenSPARK.

While the executable part of SPARK is a strict subset of Ada, SPARK also contains a contract language. These contracts allow specifications to be expressed in terms of information and data dependency, pre conditions and post conditions. Static checks, loop invariants and assertions are also supported. The logic used to express conditions is full first order classical logic with typed terms including integers, reals, arrays, records and user defined functions. Various forms of abstraction are supported including predicates and proof-only functions, separate subprogram specification and implementation, multiple orthogonal sets of contracts and abstract state variables.

By restricting the language, many verification tasks are simplified or rendered tractable. Aliasing between variables is not possible with SPARK as the language specification forbids it and the tools reject any program where aliasing occurs. The amount of memory required can be tightly bounded thus there is no risk of running out of memory on embedded controllers. Likewise removing recursion means that the stack space required is necessarily finite and can be easily computed. In many cases runtime can be bounded and many common concurrent programming problems such as deadlock are eliminated or can be reliably detected. One key property that aids verification is a strong guarantee on modularity. It is possible to analyse sub-programs (functions and procedures) using only their implementation and the specification files of any sub-programs they call. Thus a sub-program can be analysed as soon as it has been implemented and properties proven long before the program can be run. The emphasis on

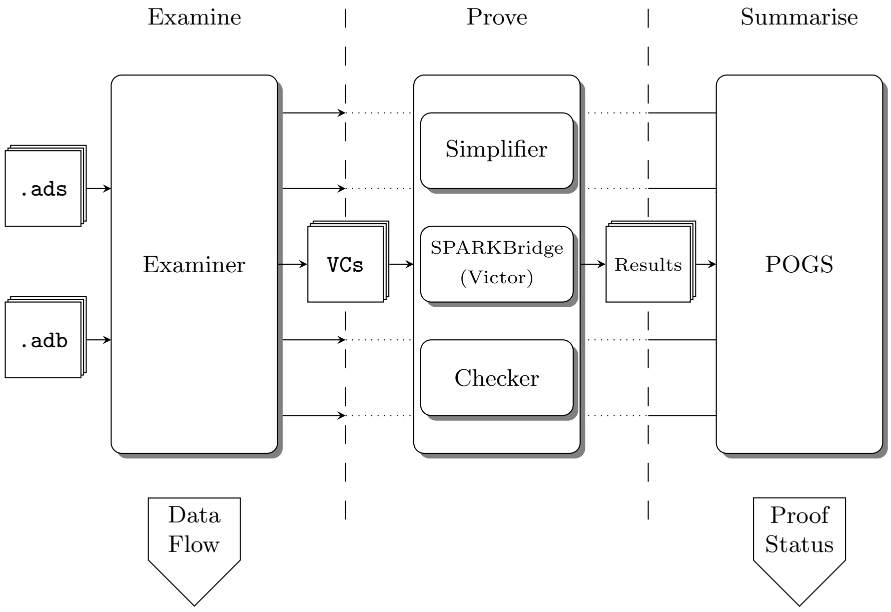


Fig. 1. The SPARK Tool Chain

modularity turns out to have other unexpected benefits as it is key to enabling the techniques described in this paper.

Language support for verification is of little use without tool support; the next section describes the SPARK static analysis tool chain.

2.2 The SPARK Tool Chain

The SPARK analysis tools are one of the most established verification suites in active use. The first version was released in March 1987. The GPL 2011 version was released in July 2011 and the next release of the commercial version, SPARK Pro [1] is due in Q2 2012. Figure 1 illustrates the current architecture of the tool chain, with the phases of computation, flow of information and outputs. There are three key phases, referred to as examine, prove and summarise.

Examine. The front-end to the system is the Examiner, this takes a set of Ada specification (`.ads`) and implementation (`.adb`) files[1] and performs the initial analysis. After parsing the input, basic syntactic checks and type checking are performed. These include checking for Ada features that are not permitted in SPARK. Next, path insensitive [12] data flow analysis is performed using Bergeretti and Carré's algorithm [3]. The results are checked against contracts giving

[1] Roughly analogous to `.h` and `.c` files although they are fully integrated into the language and have strong conditions on what each can contain.

the specified information flow, allowing violations of information separation (often important in a safety case) and redundant parameters (often indicative of bugs or specification flaws) to be detected. Data flow is the minimum analysis performed by the tools, but in the common case verification condition (VC) generation is also performed using Bergeretti's method [4]. Proof of these VCs shows partial correctness in addition to absence of all run time exceptions including numerical overflow and underflow, array bounds errors and divide by zero. The Examiner generates one `.vcg` file[2] per sub-program, each containing one or more VCs per path. Within each individual VC there are multiple conclusions, one for each of the conditions that need to be proven in the simplification phase.

Prove. The proof phase takes the set of `.vcg` files generated by the Examiner and uses a series of tools to prove the VCs they contain. The output of these tools differ, but is commonly a set of results and a log file. Although these will then be used in the summary phase, the outputs are of independent interest as they may be deliverables used as part of a system's assurance case. It is also important to note that each `.vcg` file is handled separately as the modularity guarantees of SPARK mean that they are independent of each other.

The first tool used is the Simplifier. This is a rewriting proof system implemented in Prolog. It is the fastest of the tools and the simplest. However as it is heavily tuned to the type of formulae produced by the Examiner it is not uncommon for it to discharge over 98% of conclusions showing absence of run time exceptions in well written[3] code. Although less comprehensive than various model generation tools, there are a number of advantages of using syntactic techniques. Firstly the output includes a human readable and machine checkable proof and secondly when it is not capable of discharging a conclusion it outputs a simplified version giving progress so far. The Simplifier also supports user supplied rewrite rules which allow coverage to be extended to near 100% with some manual effort.

The second tool used is Victor [22] (which is part of SPARKBridge[4]), a translator to SMTLib [31]. This converts each VC within a `.vcg` file into a formula and uses an SMT solver to attempt to prove it. Alt-Ergo [9] is the currently supported solver although CVC3 [11], Yices [13] and Z3 [29] can also be used as they can discharge AUFNIRA theories. Although Victor can prove a wider range of VCs than the Simplifier, its run-time is less predictable and in the cases where it is not able to discharge a VC it does not produce any indication of which are the problematic sub-formulae. Victor is currently regarded as experimental.

The final tool included with the SPARK tool chain is the checker. This is an interactive theorem prover that can be used to construct proofs for VCs outside

[2] Two more files are also generated: The `.fdl` file containing declarations and the `.rls` file containing substitution rules and similar. The set of these three files form the "VCs".

[3] Typically a discharge rate of less than 90% indicates badly written code or incorrect tool configuration.

[4] SPARKBridge is the umbrella term for interfacing with other proof engines besides the Simplifier.

of the capability of the automatic tools. It can also be used to check the proof logs generated by the Simplifier. Although very powerful, the time and skill required to use an interactive theorem prover and the need to maintain program and proof together means that its use is often only commercially viable on a few core sub-programs.

In addition to the three core tools there is also a counter example generator and programmer support tool (Riposte [7]) under development and a third party tool that integrates with Isabelle [5].

Since SPARK 6 (released in 2001) an additional tool, sparksimp, has been provided. This forms a list of all of the `.vcg` files in a project, optionally sorts them by size (using file size as an approximation of difficulty) and then splits the simplification over a user specified number of concurrent tasks. Each task checks if the `.vcg` file is newer than the results file (a sufficient but not optimal condition for needing simplification) and then calls the Simplifier, with Victor being optionally used on any simplified conclusions that remain unproven. Although simple, the support for incremental and parallel simplification are of considerable use, especially during development.

Summarise. Once the proof phase has been completed for all VCs, the POGS tool is run to summarise the state of the system-wide verification proof. It draws information from the results and log files produced by the proof tools and presents it in a developer friendly format. Sub-programs with failed or undischarged verification conditions are highlighted and a wide array of statistics are presented.

2.3 Verification in Practice

To give some qualitative idea of how the tools are used and the scale of the verification task, it is worth considering some statistics on the code base used in Section 4's experiments. It is a monitoring system used to support the safe operation of a piece of critical infrastructure. Multiple concurrent tasks (implemented in RavenSPARK) are necessary to handle communication with various inputs and outputs systems and achieve the hard realtime requirements. Figure 2 gives an approximate count of lines of code and breakdown.

Central to the system's safety case is the use of the SPARK tool chain to perform data flow analysis and prove the absence of run time exceptions across the whole system. Pre and post conditions on a few critical subroutines are also proven. Doing so generates approximately 12,500 `.vcg` files, containing 126,000 VCs comprising 130,000 individual conclusions. Figure 3(a) shows the distribution of the number of VCs per `.vcg` file, Figure 3(c) gives the distribution of the number of conclusions per VC and Figure 3(b) gives the distribution of the time taken to simplify each `.vcg` file. All of these graphs are plotted on a log scale showing a rough power law distribution of all of these quantities – most `.vcg` files contain only a handful of VCs each with relatively few conclusions which are trivial to prove. The size of a `.vcg` file is generally a good estimate of the time required to prove it. However there are exceptions – a small but very complex `.vcg` file can take much longer to simplify than a huge but simple one.

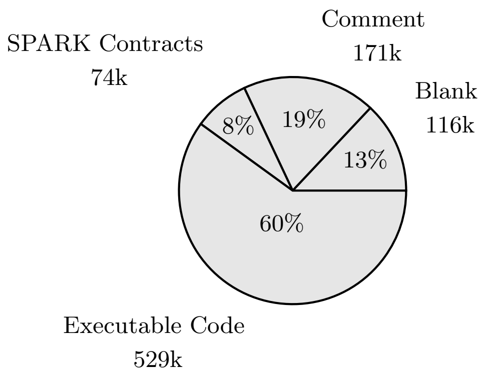
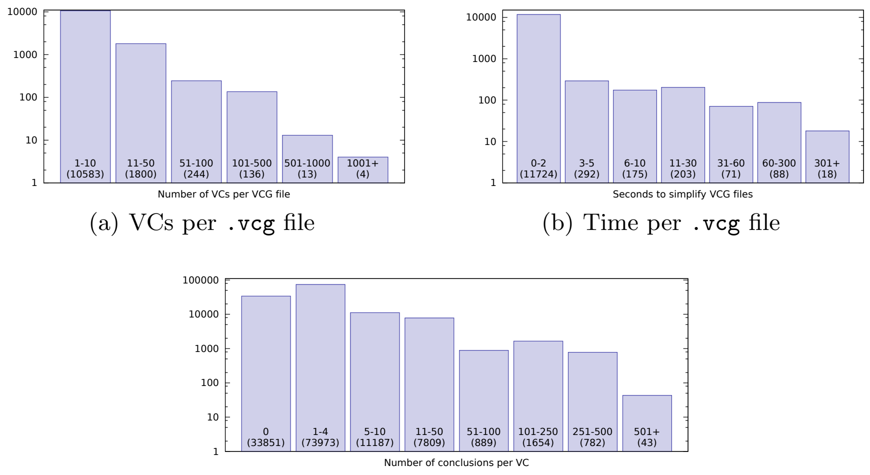


Fig. 2. Project size in physical lines of code (counted with `wc -l`). The executable code corresponds to 260,000 logical lines of code (counted with GNAT metric).



(a) VCs per `.vcg` file

(b) Time per `.vcg` file

(c) Conclusions per VC

Fig. 3. Statistics on the VCs generated

Table 2. Verification times across a number of platforms. Times are rounded to the nearest 30 seconds. The number after the slash is the number of processor cores used.

	Desktop			Server	
Processor	E6550 2.33 GHz		i7 860 2.8 GHz	2x E5430 2.66 GHz	2x X5550 2.67GHz
OS	Windows XP	GNU/Linux Debian 6.0	GNU/Linux Debian 5.0	Windows Server 2003 R2	GNU/Linux Ubuntu 10.04
Examine	79m / 1	10.5m / 2	5.5m / 4	71m / 1	4m / 8
Simplify	505m / 2	433m / 2	169.5m / 4	132m / 8	95.5m / 8
Summarise	29m / 1	10m / 1	3.5m / 1	18.5m / 1	4m / 1
Total	10h 13m	7h 33.5m	2h 58.5m	3h 41.5m	1h 43.5m

Table 2 gives timing results for the three phases of verification (note that the project does not use Victor or the checker, thus the proof phase consists out of only running the Simplifier) across a range of hardware and operating systems. The numbers given after the times are the number of threads used. Although the Examiner is not natively multi-threaded, the modularity of SPARK allows the examination task to be decomposed, thus allowing parallel analysis. In all cases the bottleneck are the CPU resources available; memory, disk and I/O performance are not limiting factors and are thus not recorded. Victor is not used by this project; if it was, some of the user written rewrite rules could be removed but at additional computational cost. Using the qualitative time band given in the introduction, verification times vary between the upper limit of "Nightly" and down towards the middle of "Daily". However all of them are firmly in the "Asynchronous" time bands. This requires the developer's day to day work to be reorganised around the tools and results are usually obtained after the changes being checked in have passed out of the developer's short term memory.

3 Increasing Verification Throughput

To reduce the time taken for verification to the "Lunch" or "Coffee" time band, a classical approach to increasing system throughput [19] is used. First, focus on the largest component of run-time; in this case the proof of VCs using the Simplifier. Next identify independent sub-tasks; in this case VCs. Caching and parallelism can then be used to increase the throughput of completed sub-tasks.

This approach raises a number of questions. Firstly, what granularity of sub-task should be used? If the subtasks are too large then parallelism may be limited and the total time cannot be reduced below the time taken to complete the longest subtask. At the other end of the spectrum, if the tasks are too small, the communications and synchronisation overhead will swamp any speedup. In the proof phase there are three obvious levels of division into sub-tasks; per `.vcg` file, per VC and per conclusion. Given the statistics in Figure 3, per `.vcg` file is a good approach: This gives enough tasks that nodes will likely not be idle but not so many that communications becomes a bottleneck. Also the tasks are generally sufficiently short to not limit the system speed up. The other alternatives will be discussed as future work.

The second question is how this integrates with sparksimp, which already implements parallel dispatch and incremental computation. The parallelism in sparksimp works at a per `.vcg` file scale, thus it is possible to fit 'under' this by wrapping the calls to the actual proof tools. Using timestamps as a sufficient condition works well when refining the implementation or proof of a particular package. However for many code changes the effects are not localised and it is necessary to run the entire examination phase again, thus updating all timestamps. Although the method described below is entirely compatible with timestamp based recomputation, in practice using timestamps and caching seem to be largely orthogonal.

At the heart of the technique for implementing distributed and incremental automatic proof is a slightly unconventional use of memcached.

3.1 Memcached

Memcached [16] is an in-memory cache originally designed to speed up dynamic web sites by caching common database queries and thus avoiding the need to access information on disk. A simple plain text protocol runs directly over a TCP or UDP connection presenting the interface of an associative array. When the configured memory limit is encountered the least recently used elements are evicted from the cache[5]. Concurrent access is handled within the daemon and multiple worker threads can be used. The simplicity of memcached allows very high performance implementation. Distribution over multiple machines to increase throughput and capacity is also supported. Furthermore, its widespread adoption means that a range of monitoring and analysis tools are available.

3.2 Integrating Memcached into the SPARK Toolchain

Adding memcached support to the SPARK toolchain is simple. A wrapper for the Simplifier hashes the `.vcg`, `.fdl` and `.rls` files the Examiner generates to produce a 160 bit SHA1 hash, `A`, and the user supplied rewrite rules[6] to produce another 160 bit SHA1 hash, `B`. The full key is then composed out of the two to give `A:B` and the wrapper checks whether this key appears in the associative array. If it does then the contents are the compressed[7] simplified `.vcg` file and this can be output directly. If not then the `.vcg` file is simplified, compressed and then stored. Line numbers are replaced with symbolic variables[8] to avoid cosmetic code changes causing cache misses. The implementation of this wrapper is only around 250 lines of Python. Wrapping other proof tools would be just as simple.

A key question about the use of memcached is to whether it could affect the assurance of the proof produced by the toolchain. Memcached is a simple, robust and widely deployed piece of software. The use of SHA1 hashes mean the chance of collision is low; POGS would also flag up any accidental collisions as the subprogram name or number of VCs of the simplified `.vcg` file would likely not match with original `.vcg` file. In all of the experiments run, cache evictions do not occur. However if they do they will only affect performance, not the correctness of the proof. If desired the logs, including the Simplifier generated proofs, can also be stored in the cache. Finally, caching can be used in development but disabled on the final production builds of the system and proof.

[5] There are protocol-compatible implementations that use a persistent backing store on disk and thus operate as associative databases rather than caches.

[6] Although these sets of rules change relatively rarely they can affect the soundness and completeness of results. Thus changes to them need to give a different hash key.

[7] We have used bzip2, but any compression scheme would work.

[8] After implementing this, the authors discovered that [23] suggests this as a technique for making caching more robust.

The cache can easily be shared between multiple developers meaning that simplified/proven VCs may already be present in the cache before other developers receive the updated code. The memory requirements for most verification runs are modest; all of the experiments performed in Section 4 use a 512 MB cache and do not require any evictions.

Distributed verification is also supported at little extra cost as; a number of machines use the same cache simultaneously. To support this, two options were added to sparksimp: A 'random shuffle' method simplifies the `.vcg` files in a random order on each node. An 'equal shares' option requires the total number of nodes, n, and i, a unique id between 1 and n for each node. Sparksimp then sorts `.vcg` files by size and simplifies every n-th one, starting at the i-th. On reaching the end of the list it simplifies the remaining VCs in a random order. The advantage of the random ordering that there is no synchronisation required between nodes. Conversely, the equal shares option gives higher throughput but it requires minimal coordination when starting the verification process.

Both approaches are also heterogeneous, ad-hoc and dynamic; if a verification run is taking too long, other machines using the 'random shuffle' approach can be started on the same task and will result in a speedup. Furthermore both distributed approaches can be combined with the incremental usage of the cache.

The use of memcached is a simple idea and easily implemented; it is a purely pragmatic approach. However, as the results in the next section show, it is remarkably effective and gives a qualitative shift in how the tools are used.

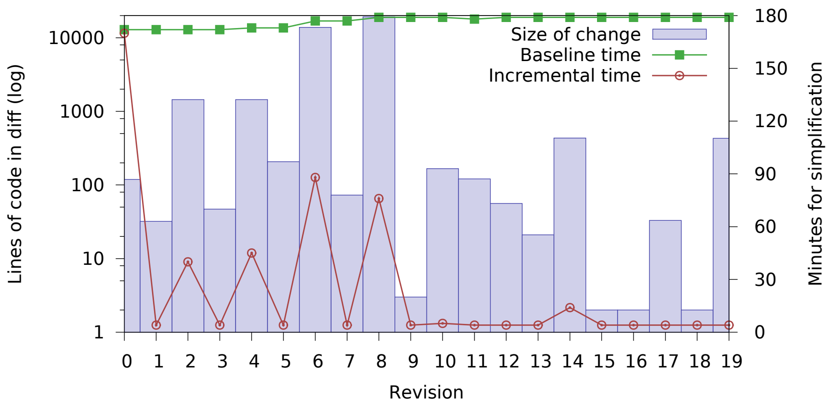
4 Experiments

To demonstrate the possible usage scenarios of a memcached-based cache and to measure their effectiveness, a series of experiments were conducted. A set of twenty distinct, real-world modifications was used. These were changes that had been developed on branches, reviewed and signed off. Thus they are larger than typical development commits.

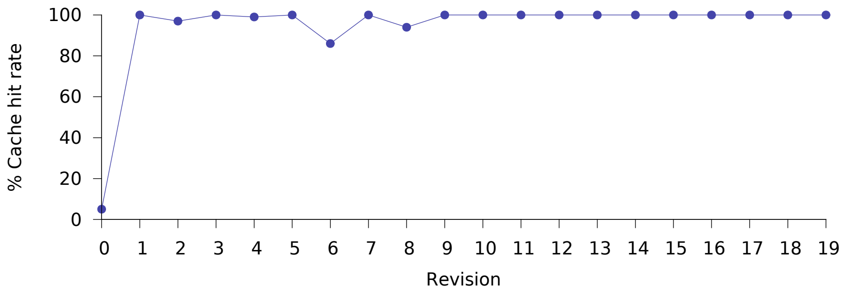
Experiments were run on desktop class computers running Debian 5.0 (as shown in Table 2) with a single 4-core 2.8 GHz Intel Core i7 860 processor. memcached was run on a separate server class computer using 512Mib of memory. The load on this machine was negligible and no cache evictions occurred. The computers were connected with 100Mb Ethernet running over commodity switches.

4.1 Incremental Solving

To test the incremental use of caching, the cache was emptied and each of the twenty commits were simplified sequentially, using the populated cache. sparksimp used four concurrent tasks to simplify the `.vcg` files in parallel. Figure 4 gives the results of this experiment. The times taken for incremental and non-incremental (baseline) simplification are given in Figure 4(a), overlayed on the size of the diff. Cache hit rates are given in Figure 4(b).



(a) Simplification time and diff size



(b) Cache hits

Fig. 4. Results of the incremental usage of memcached

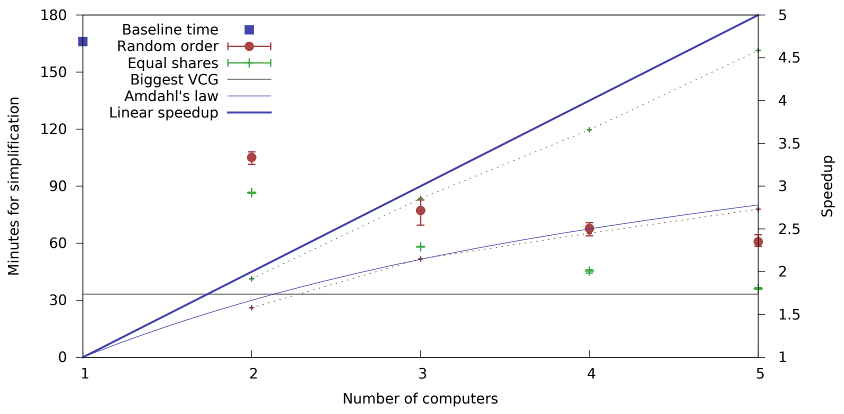


Fig. 5. Results of the distributed usage of memcached

Fifteen of the revisions were reduced to 10% or less of the original time taken by non-incremental verification, two were reduced to 25% and even the largest commits, that changed over 1% of the system's code, were at least halved. The cache miss figures show that the longer runtimes were due to a few difficult VCs rather than a large number of `.vcg` files needing to be simplified.

The 5% cache-hit rate on the initial revision is not due to duplicate code as such, but due to duplicate VCs. The project contains two sets of SPARK contracts (each with a different focus and purpose) which are applied to different parts of the system. Some packages - which sit on the boundary - contain both sets of contracts and thus identical VCs are generated for some of them.

4.2 Distributed Solving

To test the distributed case, a single version of the code base was used. The cache was emptied and a number of machines were used to simplify the set of generated VCs, but sharing a cache. We have tested both approaches for dividing up work between machines introduced in Section 3.2: 'random shuffle' and 'equal shares'.

In each test, all machine were started together and the time taken for the first computer to generate a complete set of simplified VCs was recorded. The minimum, maximum and average across three runs is given in Figure 5. On the same graph the time taken for the slowest single `.vcg` file is plotted, along with speedup per machine (dashed lines). Finally, an estimate of the optimal speedup for each approach is plotted. For the 'random' approach this is computed using Amdahl's law, using the most time-consuming `.vcg` as the sequential part of the computation. Although this is an oversimplification it gives a good estimate of what is possible. The optimal speedup for the 'equal shares' approach is linear.

Both approaches are close to their optimal performance. The 'equal shares' approach produces very predictable and stable results. With five machines used, analysis time was reduced from around 2:45 hours to around 35 minutes.

As expected, some variance in run-time is apparent for the 'random' approach, but as more computers are used results becomes more predictable. As the largest VC takes more than 30 minutes to discharge and it is dis-proportionally large, having a good result depends on this VC being analysed early by one computer and very late by the others (in order to use the result in the cache). As outlined previously, the 'random' approach takes less time to set up as the number of machines used does not have to be known in advance.

5 Related Work

Distribution and incremental computation are widely applicable techniques for increasing system performance so there are many uses of these ideas within the verification literature. Thus it is necessary to be clear on how, where and why systems use these techniques.

Most of the systems that use contracts and VC generation and are technically comparable to SPARK are research projects. There are a number of tools which implement parts of the JML standard [26] including ESC4 [24], which will be discussed below. Many of the other research projects are centred around intermediate verification languages such as Why [15] and Boogie [25]. These provide a simplified procedural language, a contract language for specifying pre and post conditions and tools for generating and proving verification conditions. Front-ends are then constructed for a variety of programming languages; examples include VCC [8], Dafny [27] for Boogie and the Jessie plug-in [30] of Frama-C, Krakatoa [15] and Hi-Lite [18] for Why.

Out of all of these systems, only the JML analyser ESC4 [24] makes use of caching and distribution. Each VC is checked against a persistent local hash table before proof is attempted. If it is already contained in the hash then the result can be returned immediately. JML4 attempts to verify each method as it is saved and caching is needed to help maintain interactivity. This is similar to the argument made in this paper, although on a smaller scale. A key technical difference is that the ESC4 cache is local and not shared between users. However when it comes to support for distributed processing there are larger differences between the two systems. ESC4 uses distribution to increase the range of propositions that can be proven by running a variety of solvers as a portfolio, thus targeting the latency of proving VCs not the throughput. The implementation of distribution in ESC4 is separate from the implementation of caching. In contrast, using memcached gives support for distributed processing at no extra cost and allows much more dynamic and ad-hoc use of multiple computers.

Recent development versions of the Why system have included an IDE which stores proof attempts in a local XML database [6]. Although this feature is aimed towards improving the usability of Why for manual theorem proving it doubles as a local, unshared cache in a similar fashion to ESC4.

Looking beyond verification conditions, incremental and distributed techniques have been used in various other verification tools. In model checking, distribution across multiple machines was originally used as a way of mitigating the high memory requirements of explicit state model checking. More recently it has been used in symbolic and CEGAR model checkers to improve performance [28]. Incremental computation and caching has received less attention in model checking. However there are algorithms for updating abstractions [20] and explicit state models [10] for similar use-cases to this paper. There is also widespread use of incremental computation in lighter weight static analysis systems [14].

Finally, the use of memcached as a shared, persistent space is somewhat reminiscent of tuple space programming models as pioneered by Linda [17].

6 Conclusion

In an industrial context, verification time is not just a number, it has a significant qualitative effect on how verification tools are used and thus on project management. Adding support for memcached and creating a shared, persistent cache of proven results is a trivial modification. The prototype implementation for the SPARK tool chain required only around 250 lines of code, implementation in other systems should be similar. However the results in Section 4 show that it has a significant impact. When used to support incremental verification from a single machine it improves runtime 29 fold (on average). When used for distributed simplification it yields near optimal speedup. More importantly this shifts verification from the upper end of the "asynchronous" time bands to the lower end of the "synchronous" time bands with a corresponding reduction in costs and increased practicality. Given its low cost of implementation and qualitative impact on the development process, the use of a persistent shared cache (such as memcached) is an obvious feature for any verification system intended to be used industrially.

A number of future developments are possible. Adding support for parallelism at the per-VC level would help mitigate the impact of the few `.vcg` files with long verification times. Likewise, adding a 'pending' flag to the cache could reduce the impact of two nodes attempting to simplify the same `.vcg` file at the same time. There are also a number of techniques that could be used to increase hit rates including caching per VC and abstracting out procedure names. Ideally the caching system could be modified to be robust against the addition and deletion of irrelevant hypotheses, although this would require integrating the caching and reasoning logic.

Although these additions will give pragmatic gains, the primary motivation of this work was to change the qualitative time band of verification and these changes are unlikely to be able to reduce verification to the interactive time band. To achieve this it will be necessary to consider other parts of the verification process. Utilising the incremental nature of software development to speed up these phases represents an overarching future challenge.

References

1. Altran Praxis: SPARK Pro. (2009), `http://www.adacore.com/home/products/sparkpro`
2. Barnes, J.: High Integrity Software - The SPARK Approach to Saftey and Security, 2nd edn. Addison Wesley (2006)
3. Bergeretti, Carré.: Information-flow and data-flow analysis of while programs. ACM Transactions on Programming Languages and Systems 7, 37–61 (1985)
4. Bergeretti, J.F.: An algebraic approach to program analysis: Foundations of a practical analysis system. Ph.D. thesis, University of Southampton, Faculty of Engineering and Applied Science, Department of Electronics (1979)
5. Berghofer, S.: Verification of Dependable Software using SPARK and Isabelle. In: Brauer, J., Roveri, M., Tews, H. (eds.) Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011). pp. 48–65. TU Dresden (August 2011); technical report TUDIFI11
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Rustan, et al. [32], pp. 53–64
7. Brain, M., Schanda, F.: The Riposte counter example generator (2011), `http://forge.open-do.org/projects/riposte`

8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
9. Conchon, S., Contejean, E., Kanig, J.: Ergo: A theorem prover for polymorphic first-order logic modulo theories (2006), http://ergo.lri.fr/papers/ergo.ps
10. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental Algorithms for Inter-Procedural Analysis of Safety Properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer, Heidelberg (2005)
11. CVC3: An automatic theorem prover for Satisfiability Modulo Theories (SMT) (2006), http://www.cs.nyu.edu/acsys/cvc3
12. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
13. Dutertre, B., de Moura, L.: The YICES SMT Solver (2006), http://yices.csl.sri.com/tool-paper.pdf
14. Eichberg, M., Kahl, M., Saha, D., Mezini, M., Ostermann, K.: Automatic Incrementalization of Prolog Based Static Analyses. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 109–123. Springer, Heidelberg (2006)
15. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
16. Fitzpatrick, B., et al.: Memcached - a distributed memory object caching system (2003), http://memcached.org
17. Gelernter, D., Carriero, N., Chandran, S., Chang, S.: Parallel programming in Linda. In: ICPP, pp. 255–263 (1985)
18. Guitton, J., Kanig, J., Moy, Y.: Why hi-lite ada? In: Rustan, et al. [32], pp. 27–39
19. Hennessy, J.L., Patterson, D.: Computer Architecture, A Quantitative Approach, 4th edn. Morgan Kaufmann (2007)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme Model Checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
21. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. 39, 92–106 (2004), http://doi.acm.org/10.1145/1052883.1052895
22. Jackson, P.B., Passmore, G.O.: Proving SPARK Verification Conditions with SMT solvers (December 2009), http://homepages.inf.ed.ac.uk/pbj/papers/vct-dec09-draft.pdf
23. James, P.R., Chalin, P.: Esc4: A modern caching ESC for Java. In: Huisman, M. (ed.) Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems, pp. 19–26. Association for Computing Machinery (2009)
24. James, P.R., Chalin, P.: Faster and more complete extended static checking for the java modeling language. Journal of Automated Reasoning 44(1-2), 145–174 (2010)
25. Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code using SMT Solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
26. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml: a behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes 31, 1–38 (2006), http://doi.acm.org/10.1145/1127878.1127884

27. Leino, K.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
28. Lopes, N.P., Rybalchenko, A.: Distributed and Predictable Software Model Checking. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 340–355. Springer, Heidelberg (2011)
29. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
30. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. Ph.D. thesis, Université Paris-Sud (January 2009)
31. Ranise, S., Tinelli, C.: The SMT-LIB format: An initial proposal. In: Workshop on Pragmatics of Decision Procedures in Automated Reasoning (2003)
32. Rustan, K., Leino, M., Moskal, M. (eds.): First International Workshop on Intermediate Verification Languages (August 2011)

A Comparison of Intermediate Verification Languages: Boogie and Sireum/Pilar

Loren Segal[1] and Patrice Chalin[2]

[1] Dependable Software Research Group, Concordia University, Montreal
[2] Kansas State University, Manhattan, Kansas

Abstract. Use of contract-based specification languages is slowly increasing. This advancement has been due in part to the growing efficiency and usefulness of Intermediate Verification Languages (IVLs) which abstract the low level details of program verification and act as a backbone for higher level tools. This paper looks at two mature IVLs, Boogie and Sireum/Pilar, and provides a comparative study of their features in order to offer resources for tool developers and IVL designers. As validation for this comparison, we introduce two tools, *ruby2boogie* and *ruby2pilar*, to illustrate the translation from Ruby to Boogie and Pilar.

1 Introduction

Though very slowly, contract-based specification languages (or language extensions) are increasingly making their way into developers' toolboxes—e.g., at Microsoft [15] and Google [9]. This is due, in part, to the availability of specification processing tools that are easy to use (i.e., automated), useful and efficient enough to make it realistic to integrate them into software development process baselines. Development of such tools, like for most other "high-integrity" software, takes considerable effort and offers serious engineering challenges. To help mitigate development costs, most tooling products and recent product families are conceptually organized following a pipe-and-filter (dataflow) architecture making a judicious use of Intermediate Verification Languages (IVLs). This leaves developers re-engineering old verification tools or creating new tools (e.g., to add support for a new language) with an opportunity and a problem: given various, potentially equivalent and suitable IVLs, which one should be adopted?

This paper takes a first step in addressing this important question by comparing two IVLs: Boogie 2 from the Microsoft RiSE Group [13], and Sireum/Pilar (Pilar for short) from the Kansas State University SAnToS Laboratory's Sireum framework [17,16]. Specifically, our **primary contributions** are:

– After establishing the scope of the comparison and offering a very brief introduction to both languages (Section 2), we present a detailed *point-wise, theoretical comparison* of the language features of Boogie and Pilar (Section 3).
– A *practical tool-developer oriented comparison* (Section 4) via the description of our experiences in developing two tools that perform automatic

source translation from Ruby [8] to Boogie and Pilar named, respectively, `ruby2boogie` and `ruby2pilar`. We use these tools to illustrate pragmatic differences in performing translation from a source language to each IVL. We believe that this comparison and validation will be useful not only to tool developers, but also IVL designers. Since neither Boogie nor Pilar were designed with a dynamic language like Ruby in mind as a source language, we believe that this makes an interesting case for the challenges faced by verification tool developers adding support for new languages.

To our knowledge, this is the first such comparison.

The work reported here is part of a larger effort to provide redesigned verification-condition (VCgen) based checkers for Java and SPARK/Ada, hence this study to determine which IVL would be most suitable for such a task. Due to the authors' appreciation for dynamic languages like JavaScript and Ruby, and the first author's extensive experience in the use of Ruby, our study ended up taking the form of the exploratory development of a static checking, symbolic execution and test case generation tool for an important subset of Ruby. As such, a **secondary contribution** of this paper is the presentation of preliminary work on a unique approach to the static analysis of Ruby programs, which offers special challenges due to its dynamic nature [19]. A summary of the comparison is offered in Section 5. Related work, conclusions and future work are covered in Sections Section 6 and Section 7.

2 Background

Intermediate Verification Languages (IVLs) exist as a way to encode computer programs into a common language while maintaining (only) the important logical and stateful properties of the original program. This common language can therefore sit between the source language and the language of theorem provers like Simplify, Z3, and Isabelle, in order to provide a higher level representation that is both more human-readable and more easily encodable from a given source language.

For example, the languages Java and C++ have very many semantic and syntactic similarities, but at the same time, also have quite a number of semantic and syntactic differences. For a developer to translate either language into verification conditions (VCs) for a theorem prover to parse, the developer would be required to implement two very distinct algorithms to perform such a translation, and the translations involved would not be trivial. Making use of an intermediate language allows developers to implement a very simple translation from either language into a common syntax, at which point a more refined tool could perform the more complex translations in converting the IVL into VCs, discharging the VCs, and performing the verification. A powerful IVL would be one that allows differing syntaxes to be encoded with the least amount of effort while still being expressive enough to ensure that the semantics of each language are preserved.

In this sense, an IVL is a great asset to developers who need robust verification tools implemented for a wide range of languages. However, it is ultimately not trivial to define a *common* intermediate language that can still support the syntax and semantics of many source languages. We look at Boogie 2 and Pilar, two possible IVLs, and compare them in terms of their functionality and effectiveness at specifying "common" programs. Although our research focuses primarily on object oriented programs, we also look at how other language paradigms might be encoded (specifically, procedural and functional) using these IVLs.

Boogie 2 [13][1] (formerly *BoogiePL* [2]) and Pilar [17] can be defined as IVLs, as they both attempt to abstract programs into simple logical steps meant for formal verification tools. Boogie is developed by the RiSE group at Microsoft Research. The Boogie verification language, as well as its open-source tool by the same name, were initially created to target the C# language through the Spec# project, though they were eventually used to support a host of other languages (including C) as well.

Pilar, a component of the open-source Sireum [16] framework, is developed by the SAnToS Lab members (Kansas State University). This language was initially used to describe both Java and SPARK programs in order for the Sireum framework to perform various analyses. Pilar is effectively the intermediate representation for all data and programs processed by the Sireum framework, and as new languages are supported by the framework, they too will be translated into this representation.

2.1 Scope of Comparison

Both Boogie and Pilar have language features and use-cases that exceed the scope of this comparison. For instance, Boogie has many features (such as axioms) that are useful when inputting manual proofs rather than performing a direct source language translation with a tool. Furthermore, Pilar is not *only* used as an IVL, and in fact is more generally intended as a "modeling language" for many forms of data analysis and model checking. We therefore focus our comparison on features of either language that assist with *automatic* translation of a source language into Boogie or Pilar as IVLs, since this is the most common use case for both languages.

2.2 Translation of a Simple Java Program

Before looking closely at the features of Pilar and Boogie we present a translation of a simple Java stack class (Figure 1) to both Boogie and Pilar so that we can become more familiar with the syntax. The stack implementation uses the Java Modeling Language (JML) to express contracts. JML is a notation for specifying and describing the detailed design and implementation of Java modules [12]. It is a Behavioral Interface Specification Language (BISL) offering, in particular, method specification by pre- and postcondition and class invariants to document

[1] From this point on, an unqualified use of the term "Boogie" shall refer to "Boogie 2".

```
public class Stack {
  private int size, arr[];

  //@ modifies size, arr;
  public Stack() {
    arr = new int[100];
    size = 0;
  }
  //@ modifies size, arr;
  //@ requires size < 100;
  //@ ensures size == \old(size)+1;
  //@ ensures arr[\old(size)]==val;
  public void push(int val) {
    arr[size++] = val;
  }
  //@ modifies size, arr;
  //@ requires size > 0;
  //@ ensures size == \old(size)-1;
  public int pop() {
    return arr[--size];
  }

  public static void main(String[] args){
    Stack stack = new Stack();
    stack.push(1);
    int x = stack.pop();
    x += stack.pop(); // error!
  }
}
```

Fig. 1. Java Stack Example (with Contracts Written in JML)

module behavior. The Pilar and Boogie representations can be found in Figure 2. It should be noted that these are manual, not automatic, translations, and may have shortcuts that are not taken when performing automatic source translation.

3 Comparison of Language Features

3.1 Basic Assertion Language

Both Boogie and Pilar have similar assertion languages that can be used to encode verification conditions and be sent to various theorem provers to verify program input. The basic assertion language used in both IVLs can be defined by the simple commands **assert**(*expr*) and **assume**(*expr*). These commands assert the validity of an expression or assume the validity of an expression respectively. The expressions themselves can be variables, simple arithmetic operators or boolean comparisons. Boogie also specifically allows two logical quantifiers, *forall* and *exists*, though Pilar allows function calls and function types as expression values, which Boogie does not.

In addition to these two basic commands, Boogie adds an extra command to this basic set known as **havoc**, which acts similarly to **assume**, encoding that some *variable* is assumed to now contain some unknown value.

3.2 Basic Control Flow

Boogie and Pilar are both, at their core, block based languages. They support control flow and branching through this fundamental concept of blocks. This allows them to model the control flow graph of source languages quite closely. It should be noted that *BoogiePL* (the original version of Boogie) was purely block based and had no abstraction for procedures, which illustrates the fundamental nature of this construct.

Location and Blocks. In Boogie, a block refers to a sequence of statements to be executed in order. Every procedure has at least one block, though if a block

```
type Ref, Type;
const Stack: Type;
var Stack.arr: [Ref][int]int;
var Stack.size: [Ref]int;

// Type hierarchy axioms omitted.

procedure Stack$ctor(this: Ref)
    modifies Stack.size[this],
             Stack.arr[this]; {
  var emptyArray: [int]int;
  Stack.arr[this] := emptyArray;
  Stack.size[this] := 0;
}
procedure Stack.push(this:Ref, val:int)
    modifies Stack.size[this],
             Stack.arr[this];
    requires Stack.size[this] < 100;
    ensures Stack.size[this] ==
      old(Stack.size[this])+1; {
  var prevNum: int;
  prevNum := Stack.size[this];
  Stack.size[this] := prevNum + 1;
  Stack.arr[this][prevNum] := val;
}
procedure Stack.pop(this:Ref)
    returns (r:int)
    modifies Stack.size[this];
    requires Stack.size[this] > 0;
    ensures Stack.size[this] ==
      old(Stack.size[this])-1; {
  var curNum: int;
  curNum := Stack.size[this];
  Stack.size[this] := curNum - 1;
  r := Stack.arr[this][curNum - 1];
}
procedure Stack.main(args: [Ref]Ref)
    modifies Stack.size;
    modifies Stack.arr; {
  var stack: Ref, x: int, y: int;
  call Stack$ctor(stack);
  call Stack.push(stack, 1);
  call x := Stack.pop(stack);
  x := x + y;
}
```

(a) Boogie

```
record Stack {
  Integer arr[];
  Integer size;
}

procedure Stack$ctor(Stack this)
    @modifies(this.size)
    @modifies(this.arr)
{
  # this.arr = [];
  # this.size = 0;
}

procedure Stack.push(Stack this,
                     Integer val)
    @modifies(this.size)
    @modifies(this.arr)
    @pre(this.size < 100)
    @post(this.size ==
      old(this.size) + 1)
{
  # this.size := this.size + 1;
  # this.arr[this.size - 1] := val;
}

procedure Integer Stack.pop(Stack this)
    @modifies(this.size)
    @modifies(this.arr)
    @pre(this.size > 0)
    @post(this.size ==
      old(this.size) - 1)
{
  # this.size := this.size - 1;
  # return this.arr[this.size];
}

procedure Stack.main(String args[]) {
  local Stack stack,
        Integer x;
  # stack.push(1);
  # x := stack.pop();
  # x := x + stack.pop();
}
```

(b) Pilar

Fig. 2. Java Stack Example Translated into Boogie and Pilar

is not specified at the start of a procedure, Boogie will create an anonymous implicit block. The Boogie procedure in Figure 3 shows two blocks, one implicit, and one explicit. The first two statements are part of the implicit block that Boogie adds to the start of the procedure and the last statement is part of the *subtractX* block. If no `goto` statement is provided for a jump, Boogie will automatically jump to the next block in sequence. Therefore, the statements 1, 2 and 3 will be executed in order.

Pilar has the same basic concept of blocks, but they are called “locations”. The equivalent of the Boogie example is shown in Figure 3(b). Pilar requires the explicit declaration of the first location, though it can be anonymously named. Finally, as shown, Pilar and Boogie will both implicitly jump to the next block (or location) in the source, if an explicit jump is not provided.

In addition to standard block sequences, Pilar also supports non-deterministic choice through a “choice operator” (explained further in the next subsection),

```
procedure run() {
  var x: int, y: int;
  x := x + 1; // #1
  y := x + y; // #2
subtractX:
  x := x - y; // #3
}
```

(a) Boogie

```
procedure run() {
  local Integer x, Integer y;
  # x := x + 1; // #1
    y := x + y; // #2
  #subtractX.
    x := x - y; // #3
}
```

(b) Pilar

Fig. 3. Locations in Boogie and Pilar

similar to Dijkstra's guarded commands [6], which potentially allows for *parallel execution* of statements. Although this feature is not discussed, it can be useful for the modeling of concurrent systems, or where there is non-deterministic behaviour. It is unclear how Boogie would be able to model similar concurrent (or non-deterministic) systems.

Branching and Looping. Both Boogie and Pilar can deal with control flow in terms of unstructured `goto` or `return` statements, which can be placed in any location or block. Boogie, however, has many convenience syntaxes for elements such as if statements and loops, and does not require `goto` statements or blocks for these. To exemplify the syntax for both languages, consider a Java for-loop with an if statement inside of it:

```
int x, r = 0;
for (x = 0; x < 10; x++) {
  if (x < 5) r = r + 1;
  else r = r + 2;
}
```

Figure 4 presents one possible encoding of such a loop into Boogie and Pilar respectively. Boogie resembles the high level Java syntax much more closely and is therefore much more convenient to encode to. Specifically, a translator would not need to keep track of (or even consider) location names as is the case for the Pilar equivalent code. Since most popular languages use structured looping and branching constructs such as if/else and for/while, this significantly simplifies translations.

3.3 Annotations

Pilar relies heavily on its annotation syntax to encode source language-specific constructs. As we will see, even contracts are specified through annotations. In this sense, annotations are a very important part of the language syntax. Even types can be encoded using annotations:

```
procedure inc(x @Type Integer) { # x := x + 1 }
```

```
var x: int, r: int;
x := 0; r := 0;
while (x < 10) {
  if (x < 5) { r := r + 1; }
  else { r := r + 2; }
  x := x + 1;
}
```

(a) Boogie

```
local Integer x, Integer r;
# x := 0; r := 0;
#loop. :: (x < 10)
     +> goto if;
     | else goto endloop;
#if. :: (x < 5)
     +> r := r + 1; goto endif;
     | else r := r + 2; goto endif;
#endif
  x := x + 1; goto loop;
#endloop
```

(b) Pilar

Fig. 4. while and if statements in Boogie and Pilar

However, it seems as though abuse of this annotation syntax can end up delegating too much of a source language's features to individual back-end tools, leading to too much complexity in the back-end tooling. For instance, encoding types as annotations entirely bypasses the inheritance and sub-typing semantics that one would get "for free" by using the `record` keyword to declare a type. It would therefore rarely be recommended to encode types in this manner in Pilar.

Boogie also allows for annotations (though they are called *tool directives*) in the form:

```
var { :NonNull } x: Ref;
```

Such a variable x would be marked as `NonNull`. The equivalent Pilar would be `MyClass x @NonNull`. Neither of these formats have any semantic meaning in the language as-is. Tools would have to look for these annotations and encode the semantic meaning themselves, either via another source transformation or a computation.

It is important to note that although Boogie has annotations, it only has them in a limited context. Specifically, they can not be defined for assignments or branch/loop syntaxes.

3.4 Specification of Contracts

Specifying Pre and Post Conditions. Boogie has a special construct for specifying pre and post conditions on a procedure. The full syntax is shown in Figure 5: multiple `requires` or `ensures` clauses are allowed, and the `old` operator refers to the state of a specific variable in the pre-state (before the procedure run).

Pilar has no set syntax for declaring such clauses. In Pilar, one would use annotations to encode pre- and post-conditions and rely on tools to process this information. For instance, if VC generation is performed, it would be the tool's responsibility to check for properly named annotations. Although this allows for more flexibility, it also requires more discipline to ensure that the

```
procedure inc(x:int) returns (r:int)
  requires x >= 0; requires x < 100;
  ensures old(x) + 1 == r;
{ r := x + 1; }
```

(a) Boogie

```
procedure inc(Integer x)
  @pre(x >= 0) @pre(x < 100)
  @post(old(x) + 1 == x)
{ # x := x + 1; # return x; }
```

(b) Pilar

Fig. 5. Pre and post conditions in Boogie and Pilar

tools used to translate the source to Pilar are compatible with the tools used to process the generated Pilar code. It would be also be the source translation tool's responsibility to insert contracts using the correctly named annotations (as expected by the rest of the tools in the workflow). Therefore, *one* possible Pilar equivalent of the Boogie example is shown beside it in Figure 5.

Specifying Loop Invariants. Loop invariants in Boogie are specified with the `invariant` keyword attached to looping constructs. Again, Pilar uses annotations to define these invariants.

3.5 Modeling Data Structures and Object Oriented Type Systems

One of the most basic features an IVL should support is the encoding of language specific data structures. Both Pilar and Boogie have relatively different syntactic methods of encoding data structures, though their semantics are roughly the same.

Pilar has a syntactic `record` element which is similar, in a sense, to C's `struct`, and is the singular method of encoding any data structure in the language. A record, like a Java class, can inherit from another and it can also be declared abstract. Boogie, on the other hand, has no construct to represent classes or data structures. It uses the `type` keyword along with `var` declarations to define symbols in a flat namespace which represent the fields. For reference, we will use the `Stack` class from Figure 2 to illustrate how data structures are modeled.

The flat namespace that Boogie uses is an important difference in the way heap-based structures are modeled between the two languages. Specifically, Boogie gives the user full control over defining how to model "memory allocation", and has no concept of object instantiation. In fact, a considerable amount of detail can be found in the Boogie manual [14] about the ways in which the heap can be encoded. On the other hand, Pilar handles allocations through a `new` keyword. Although Boogie's methodology allows for much more flexibility, it is not exactly clear which languages require this much control over heap modeling.The cost of this flexibility is complexity in modeling object-based systems. As we saw from the Boogie example in Figure 2, the declaration

```
var Stack.arrSize: [Ref]int;
```

models the field `arrSize` from class `Stack`. However, to do this in a flat namespace, the field must be translated into a variable map of references to values, where *references* are the instances of the `Stack` class, and *values* are of the type defined for `arrSize`. To reference the field data under this scheme, we write `Stack.arrSize[o]` where *o* is of type *Ref*. We must also have introduced this reference type *Ref*, whereas the Pilar code does not require defining a reference pointer type.

As we have seen, Boogie does not impose any typing rules. They may or may not be specified prior to static analysis. Again, this makes Boogie more flexible for languages with non-traditional type systems, while Pilar tends to be optimized for OO-based languages. In Figure 2 we briefly saw the translation of a `Stack` class into Boogie code, including basic type specifications. However, we did not include the specification of inheritance rules in this example. To do so, would require the use of axioms (and an extra supertype declaration) as follows:

```
const Object
axiom Stack <: Object;
```

The code above declares the class *Stack* to be a subclass of *Object*. The same semantics is implicitly defined in our Pilar example, since a record will automatically extend Object (if no explicit superclass is defined). The inheritance syntax of Pilar is like that of Java:

```
record ColorPoint extends Point { }
```

However, unlike Java, Pilar supports multiple inheritance. Note that Boogie can emulate multiple inheritance by modeling the relationships through independent axioms.

Generics Support. Boogie and Pilar both support generics (or "parameterized types") on type declarations. The syntax of this feature in each IVL is presented in the technical report [20].

3.6 Unique Features and Tool Support

Each IVL hosts a set of unique language features, which, due to space constraints, are not covered in this paper. Specifically, Boogie supports an elegant type system and the ability to define semantics through axioms and mathematical operators. Pilar supports first class functions and procedures, dynamic typing and exception handling. These features, as well as implementation considerations for tool developers, are discussed in the extended technical report version of this paper [20].

4 Preliminary Validation: Ruby to Boogie and Pilar

As a preliminary validation, we present and discuss two tools, `ruby2boogie` and `ruby2pilar`, that perform source translation from the Ruby programming language [8] to their respective IVLs. Since neither Boogie nor Pilar was designed with Ruby in mind, we believe that this comparison is an accurate representation of the challenges faced by verification tool developers adding support for new languages.

In both of these tools, we target a subset of the Ruby language for integer math, basic conditional support, and basic modeling of properties and methods on classes. This subset illustrates some, but not all, of the more interesting and practical features that are compared in Section 3. Since Ruby is dynamically typed, we rely on source code annotations embedded in comments to provide optional type information and member declarations when necessary for translation or the underlying verification tools. We also use annotations in order to provide contract specifications. The details of the supported language translations are discussed in this section.

4.1 Type System Modeling

Ruby is an object-oriented, dynamically typed programming language. Object orientation can be modeled by both IVLs, however dynamic typing presents a problem for Boogie in particular. To deal with this lack of dynamic typing support, we model Ruby's type system in `ruby2boogie` after the canonical C implementation of the Ruby interpreter known as *MRI*. Since C is a statically typed language, the implementation (and its API which allows developers to write Ruby extensions using native code) refers to all Ruby types as a `VALUE` type in C, which is a type alias for a `long`. We therefore model this similarly in Boogie, starting every translated Boogie program with (where `int` represents the mathematical integers in Boogie):

```
type VALUE = int;
```

Note that the *MRI* implementation of Ruby will reserve the first bit of an integer to denote whether the `VALUE` is an object reference or integer. Consider the Ruby method in Figure 6(a) that calculates the cube of a given integer `x`. Figure 6(b) shows the MRI native C implementation of this cube method. A similar translation to Boogie performed by `ruby2boogie` is given in Figure 7. Although we do not perform any actual type checking here, we could enforce typing the way it is done for any source language through `requires` and `ensures` contract clauses that verify type constraints. Alternative type checking strategies are still being explored and therefore not discussed, however it should be noted that implementing this type checking is more complex than the type checking implemented in `ruby2pilar`.

Pilar allows for a much less involved approach to modeling Ruby's type system, thanks to its inherent support for dynamic typing. In short, we generally need not worry about object or native types, and allow the IVL to handle these details. If type constraints can be provided in the source language via annotations, they are inserted during translation, otherwise they are simply omitted. For instance, `ruby2pilar` translates the above cube method as shown in Figure 7(b).

As mentioned previously, if a type annotation is provided in the Ruby source, it is taken as a type specification for the variable. For instance, Figure 8 illustrates

```
def cube(x)
  x * x * x
end
```

(a) Ruby

```
VALUE rb_cube(VALUE self, VALUE x) {
  long v = FIX2LONG(x);
  return LONG2FIX(v * v * v);
}
```

(b) MRI CRuby

Fig. 6. A cube method in Ruby and MRI CRuby

```
procedure #cube(self: VALUE, x: VALUE)
    returns ($result: VALUE) {
  $result := x * x * x;
  return;
}
```

(a) `ruby2boogie`

```
procedure #cube(self, x)
{
  # return x * x * x;
}
```

(b) `ruby2pilar`

Fig. 7. Translation of the cube to Boogie and Pilar

the resulting Ruby and Pilar translation if the cube method was modified to include a type annotation (the `Fixnum` class in Ruby represents the native integer type). With such an annotation, the method is now implicitly constrained to a type, and type checking can be provided lower in the pipeline (by the verification tools or Sireum itself).

4.2 Handling Arrays

One specific issue of importance to Ruby is the translation of arrays. It is intuitive in many cases to translate arrays from the source language directly as an array in the IVL, but in the case of a dynamic language, this cannot always be easily done. Once again, this issue applies more specifically to the `ruby2boogie` implementation than it does to `ruby2pilar`. This is because Boogie expects the declaration of an array to be of the form:

```
var myArr: [int]int; // map of ints -> ints
myArr[0] := 1; myArr[1] := 2; // ...
```

The type `[int]int` is a distinct type, and different from the simple type `int`. By the same token, the Ruby modeled type of `VALUE` would be different from an array `[int]VALUE`. We cannot assign one to the other, and therefore we would have to pay special attention to such variables. Fortunately, we can avoid the issue by avoiding Boogie's array syntax altogether. Instead, arrays in `ruby2boogie` are mapped as simple `VALUE` objects, and array values are populated using `assume` statements for each array element and a function that represents the array data. Pilar, on the other hand, can represent this data with the single initialization expression `arr := [1,2,3];`

4.3 Modeling Fields

Fields present an interesting problem in Ruby, because the language has no special syntax for declaring "fields" (data members of a class). Data members

```
# @param [Fixnum] x
def cube(x) x * x * x end
```

(a) Ruby code

```
record Fixnum extends Integer {}
procedure #cube(self, Fixnum x) {
  # return x * x * x;
}
```

(b) `ruby2pilar`

Fig. 8. Translation of Ruby (with a type annotation) to Pilar

are called *instance variables* in Ruby, and do not need prior declaration. To handle this, the source is parsed and every instance variable token (recognized by the syntax `@varname`) is automatically represented as a field in the model. For instance, the Ruby version of the `Stack` implementation of Figure 1 might look like (push, pop and main methods omitted):

```
class Stack
  def initialize; @arr = []; @arr_size = 0 end
end
```

Translating the source for the above Ruby code recognizes the two fields `arr` and `arr_size`. Note that we could also add type information to these declarations, or even declare new fields (that are not visible during translation) through annotations.

In the `ruby2boogie` implementation, these fields are mapped similarly to how they were illustrated in Figure 2, the main differences being that we (a) do not declare `arr` as an array type, and (b) use `VALUE` rather than `Ref` or `int`. Below is what the two fields are translated into, using the `ruby2boogie` tool:

```
var Stack$arr: [VALUE]VALUE;
var Stack$arr_size: [VALUE]VALUE;
```

The `ruby2pilar` implementation takes advantage of the Pilar `record` syntax seen in Figure 2. With no annotations, the record is simply:

```
record Stack { arr; arr_size; }
```

4.4 Control Flow

Both Boogie and Pilar support relatively high-level control flow constructs. Conditional branching using an `if` statement are supported in both IVLs (though in Pilar the "choice" `::` operator is used instead). Since Ruby has an `if` keyword, this mapping is quite straightforward.

Looping, however, is slightly more challenging. Ruby does have `for` and `while` keywords that map fairly directly to traditional looping constructs in both Boogie and Pilar, however Ruby also allows (and recommends) that looping be performed using closures as arguments to methods such as `times`, `each`, or `loop`. A traditional loop in Ruby looks more like the following:

```
total = 0
[1,2,3,4].each {|x| total += x }
```

The above would roughly translate to the following C code for the built-in version of the each method:

```
void each(int *total, int value) {  *total += value; }
void main() { int total, i, arr[] = {1,2,3,4};
  for (i = 0; i < 4; i++) each(&total, arr[i]); }
```

Therefore, translating such a loop in either IVL is a much more complex process, and there are a few ways to do it. One such method is to translate the `each` call into a traditional while or for loop, however this would only work for a specific set of built-in looping methods. Another technique is to model the Ruby's closure construct completely (which is necessary for full translation anyway) and generate separate "anonymous" procedures (or inline functions in Pilar's case) representing the closures. Note that although Pilar supports inline functions, Ruby's closures can access data and variables in the outer scope of the closure, whereas Pilar's closures cannot. This makes translation much more difficult in using either inline functions *or* anonymous procedures.

In short, it is not entirely clear which IVL is more suitable to encode looping constructs in a language with closures such as Ruby. Although it would seem that Pilar's type system would be more conducive to this type of a translation, there are many technical difficulties beyond typing that make this translation complex. It should be noted that the `ruby2boogie` and `ruby2pilar` case study tools lack any significant loop structure support for Ruby at this time due to the complexity of translating these constructs.

5 Summary

Table 1 shows a summary of the language features and tool support of Boogie and Pilar for the topics covered in this paper. The table highlights some of the main differences in these two languages, and helps us justify some of these differences. Specifically, we see that Boogie is targeted mainly to one backend (VCGen). Pilar, on the other hand has been used for different forms of verification from model checking to symbolic execution, test case generation, and most recently (and still in development), VCGen. This has affected the features that these languages target. For example, the lack of non-deterministic choice or more complete record structures for data modeling in Boogie are likely due to its focus on VCGen instead of other verification methods such as model checking. Similarly, the lack of native contract specification clauses in Pilar is due to the fact that its predecessor languages, the Bogor Intermediate Representation [18] and the Bandera Intermediate Representation [5], essentially used other constructs for program specification such as basic assertion and observable clauses. Pilar language designers have opted for support of contracts via the flexible annotation feature, pending decisions regarding the most suitable way to encode contracts—e.g. Spec# and Boogie support only simple "single-case" contracts, whereas JML supports complex contracts potentially involving multiple so-called

Table 1. A Summary of Boogie and Pilar Features

Feature	**Boogie**	**Pilar**
Language Features		
Basic statements	assert, assume, assignment, call	assert, assume, assignment, call
Control flow (basic)	if, while, goto	guards, jumps (if, case, goto), return
Control flow (exceptions)	none	throw, catch
Annotations can be written alongside . . .	declaration, assert/assume, procedure, invariant, quantifier	almost any syntactic construct
Namespace	flat	package, record
Type system	(higher-ranked) polymorphic	order-sorted
Basic types	int, bool, bit vector (bv*), polymorphic maps	numeric, string, tuple, list, set, function, relation, multi-array
Modeling data structures and Object-oriented type system	must use encoding scheme	native (support for multiple inheritance)
Literals	true, false, numeric	for all basic types
Procedural abstraction	yes	yes
Contracts, invariants	native	via annotations
Tool Support		
Front-ends / input languages	Chalice, Dafny, Havoc, Spec#, Vcc, others	Java, Spark/Ada, Ruby*
Back-ends	VCGen	Model Checker (Bogor), Symexe (Kiasan), Test case generation, VCGen*

(*) Currently under development.

"specification-cases" [12]. While multiple specification-case contracts can be expressed as single-case contracts, it is not clear that single-case contracts are the most suitable choice for an IVL.

6 Related Work

IVLs other than Boogie and Pilar exist. We first mention *FreeBoogie* [11,4] an open-source implementation of Boogie that is built on Java and therefore has superior multi-platform support to Boogie's .NET codebase. It is licensed under the MIT license, and is fairly actively developed [10].

Why, both the name of a VCGen-based verification platform and the IVL it uses, supports many of the features discussed in this paper [7]. Its tooling is built for Java and C VCGen, with back-end support for many theorem provers including *Isabelle*, for which there is currently experimental support in Boogie [3] but no support in Pilar. *Why* is published under GPLv2, making it open source and easily modifiable, similar to Pilar.

Spec# [1] is the main source language for which Boogie is targeted. The language extends the popular C# .NET language, by adding contracts and a non-null type system, among other features. The Boogie tool has native understanding of C# bytecode in order to directly communicate with the Spec#

compiler and IDE. Spec# introduces many of the features that Boogie supports, including method contracts, class invariants and field checking.

7 Conclusion and Future Work

Boogie and Pilar are both powerful tools for software verification. They are each able to encode a host of language features into their respective languages. This enables developers to take advantage of complex verification tools without having to re-engineer these tools to work with new source languages.

In our study, we compared the individual features of each IVL in order to determine which language might be more suitable in encoding certain source languages (e.g., SPARK/Ada, Java, and Ruby). We found that, although Boogie has many more high level constructs that make translation easier (e.g., branching and looping), it is less effective at handling dynamically typed or untyped languages due to the restrictiveness of its type system. We believe that Boogie would benefit greatly from alternative typing options. On the other hand, Pilar's reliance on user-defined annotations and lack of constructs for contract specification makes it difficult to encode contracts in a standardized fashion. Pilar would be more effective (especially for VCGen) if contract specification was standardized as part of the basic grammar.

To validate our claims, we implemented two proof of concept tools, `ruby2boogie`, and `ruby2pilar`. These tools perform automatic translation of Ruby to Boogie and Pilar respectively. From this exercise, we are able to qualify the pros and cons of each IVL from the point of view of their intended audience of language tool developers. We studied, in specific, the support for the modeling of object oriented paradigms, as well as the support for dynamic typing commonly found in many new languages. We plan to focus our future work on a more comprehensive study of the features of each IVL by increasing the Ruby language feature coverage in our two Ruby source translation tools. We also plan to perform a quantitative comparison of each IVLs own tooling pipeline (VC generation, for example). Finally, as a means of broadening our comparison of IVLs, we plan to include *Why* in such a study, as well as a comparison of its features in relation to the two IVLs discussed in this paper.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
2. de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.): Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, November 1-4. LNCS, vol. 4111, pp. 243–258. Springer, Heidelberg (2005)
3. Böhme, S., Moskał, M., Schulte, W., Wolff, B.: Hol-boogiean interactive prover-backend for the verifying c compiler. Journal of Automated Reasoning 44, 111–144 (2010), http://dx.doi.org/10.1007/s10817-009-9142-9

4. Chrząszcz, J., Huisman, M., Schubert, A.: BML and Related Tools. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 278–297. Springer, Heidelberg (2009)
5. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby.: Expressing checkable properties of dynamic systems: The bandera specification language. International Journal on Software Tools for Technology Transfer (STFTT) (2002)
6. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
7. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
8. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language, 1st edn. O'Reilly (2008)
9. Google Code: cofoja: Contracts for Java, http://code.google.com/p/cofoja/
10. Grigore, R.: FreeBoogie, http://code.google.com/p/freeboogie
11. Grigore, R.: Efficiency of Extended Static Checkers. Tech. rep., PhD Research Plan. UCD Dublin (December 2007)
12. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers, Boston (1999)
13. Leino, K.R.M.: This is Boogie 2. Tech. Rep. KRML 178, Microsoft Research (June 2008)
14. Leino, K.: This is Boogie 2. Manuscript KRML 178 (2008)
15. Leino, K.: Verification tools at Microsoft (January 2009); Invited talk, Digiteo seminar
16. Robby: Sireum website, http://www.sireum.org
17. Robby: Sireum: A Software Analysis Platform. SAnToS, Kansas State Univerity (February 2007)
18. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 267–276 (2003)
19. Segal, L.: Automatic program verification and test case generation of ruby programs. Tech. Rep. DSRG-TR-2011-02, Concordia University (2011)
20. Segal, L., Chalin, P.: A comparison of intermediate verification languages: Boogie and sireum/pilar. Tech. Rep. DSRG-TR-2011-01, Concordia University (2011)

LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR[⋆]

Florian Merz, Stephan Falke, and Carsten Sinz

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{florian.merz,stephan.falke,carsten.sinz}@kit.edu

Abstract. Bounded model checking (BMC) of C and C++ programs is challenging due to the complex and intricate syntax and semantics of these programming languages. The BMC tool LLBMC presented in this paper thus uses the LLVM compiler framework in order to translate C and C++ programs into LLVM's intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts as used in C and static/dynamic casts as used in C++. An empirical evaluation shows that LLBMC compares favorable to the related BMC tools CBMC and ESBMC.

1 Introduction

Bounded model checking (BMC) [3], introduced by Biere *et al.* in 1999, is a popular technique for bug finding and verification of hardware designs that is widely used in an industrial setting. For bug finding of software, BMC of C programs was introduced by Clarke *et al.* in 2004 [8], and has shown its strength in checking a variety of aspects of embedded and low-level system software (see, e.g., [16,23]). Tools implementing BMC for C programs include CBMC [8] (developed by D. Kröning *et al.*), F-Soft [15] (developed at NEC Laboratories America), SMT-CBMC [1] (developed by A. Armando *et al.*), and ESBMC [10] (developed by L. Cordeiro *et al.*).

To build a BMC tool that supports all language features of a high-level language like C or C++ reliably, including common non-standard extensions that are used by, e.g., the GCC compiler, is a daunting task. This is mostly due to the complex syntax and intricate, sometimes ambiguous, semantics of these languages. The bounded model checker LLBMC presented in this paper therefore performs BMC not on the source code level but on the level of a compiler intermediate representation (IR). This approach offers a range of advantages:

⋆ This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

- The compiler IR possesses a much simpler syntax and semantics than C/C++ and thus eases a logical encoding considerably. Furthermore, most features of C and C++ can be supported without much effort.[1]
- The program that is analyzed is much closer to the program that is actually executed on the computer since ambiguities of C/C++'s semantics have already been resolved. Furthermore, it becomes possible to find bugs introduced by the compiler.
- In producing the IR, compilers already use program optimizations that can also result in simplified BMC problems.
- The use of a compiler IR makes it possible to perform BMC on programs written in a variety of programming languages.

The use of an IR makes LLBMC, to the best of our knowledge, the *only* BMC tool that can be successfully applied to non-trivial C++ programs (CBMC contains rudimentary support for C++ but failed to analyze nearly all of the over 50 C++ programs we tried it on). A drawback of using an IR is that bugs that are (intuitively) present in the C/C++ program may be "optimized away" by the compiler (but notice that the bugs would then also not occur during execution of the program if the same compiler is used to produce the executable).

Besides using a compiler IR, LLBMC offers the following key features:

Large Set of Built-In Checks: LLBMC provides a comprehensive set of built-in checks which are described in detail in Sect. 2.

Extensive Simplification: LLBMC uses simplification techniques on different levels. First, using the optimizations of the compiler front-end generates smaller and simpler IR programs. In particular, memory-related compiler optimization techniques (e.g., moving memory operations to registers whenever possible) can simplify the BMC problem significantly. Second, rewriting techniques are used on the logical representation, e.g., to propagate constants or to simplify arithmetical and Boolean expressions.

Memory is Modeled as a Flat Byte-Array: This design decision is in contrast to what is implemented in many other tools (e.g., pre-3.9 versions of CBMC [8] or deductive verification tools such as VCC [9]) which use a typed memory model. In a typed memory model, memory is a collection of typed objects rather than a sequence of bytes. Using a byte-array makes it possible to support programs which make use of C's weak type system, e.g., by converting an `int` to a sequence of `char`s that are written to a file. Another example is the use of a `union` for the conversion between types. Typically, modeling memory on the byte level causes a performance penalty in BMC, but LLBMC uses simplification techniques that, according to an empirical evaluation, compensate for this.

Section 2 recalls BMC of software and discusses the built-in checks of LLBMC. The compiler framework LLVM is briefly introduced in Sect. 3, while Sect. 4

[1] Currently, LLBMC does not support floating-point numbers, exception handling and run-time type information (RTTI).

and 5 give details on LLBMC's approach. An empirical evaluation is presented in Sect. 6. Section 7 discusses related work and Sect. 8 concludes.

2 BMC and LLBMC's Built-In Checks

Software inherently deals with unbounded data structures such as linked lists or trees. This may give rise to infinite program runs and property checking of such programs is in general undecidable. For bug finding, BMC thus limits all program runs to finite ones, thereby achieving decidability. The bound is imposed by restricting the number of nested function calls and loop iterations that are considered. BMC performs function inlining and loop unrolling (up to these bounds), resulting in one large function that is then subject to further analysis.

LLBMC has an extensive set of built-in checks for commonly occurring bugs in C programs. Furthermore, user defined checks (specified via C's `assert` function) are supported as well. Each of these checks can be enabled or disabled independently, but most of them are enabled by default.

Arithmetic Overflow and Underflow: Arithmetic overflow[2] occurs when the result of a signed or unsigned arithmetic operation cannot be represented with the available number of bits. While the semantics of unsigned integer overflows is well-defined by the C standard using modular arithmetic, this is not true for their signed counterparts. The semantics of signed integer overflows are intentionally under-specified in the standard to give different implementations room for optimizations. Thus, any signed arithmetic overflow in a program may give rise to undefined behavior and LLBMC checks for them by default. Checks for unsigned arithmetic overflows are enabled only if requested.

Logic or Arithmetic Shift Exceeding the Bit-Width: While most programmers are familiar with arithmetic overflows, shift operations are a less well-known cause for undefined behavior. The C standard leaves shift operations like n `<<` l undefined if l is larger than or equal to the bit-width of n.[3] LLBMC supports checks for this kind of error by default since this behavior is not expected by most programmers.

Memory Access at Invalid Addresses: One of the most important classes of errors is caused by invalid memory access operations. The prime example of this are security-critical buffer overflows. An access operation for an object on the heap is only valid if it is completely contained within a block of memory which was previously allocated using `malloc`. Due to C's unrestricted pointer arithmetic, invalid memory access operations are a frequent source of crashes and vulnerabilities. LLBMC detects invalid memory accesses on the stack, on the heap, and for global variables.

[2] In the following, "overflow" is used to denote both overflow and underflow.

[3] On many architectures, shifting x by l bits is equivalent to shifting x by $l \bmod b$ bits, where b is the bit-width of x's data type.

Invalid Memory Allocation: Heap memory allocations are considered invalid by LLBMC if a memory block of the requested size can be allocated under no circumstances. Currently, LLBMC approximates this by checking if the total size of all allocated blocks would exceed the size of the heap.

Invalid Memory De-Allocation: A call to `free(p)`/`delete p` is invalid if a memory block starting at `p` was already de-allocated, was never allocated, or if `p` points to an address which is not the first byte of an allocated memory block. LLBMC checks whether either of these situations occurs.

Overlapping Memory Regions in `memcpy`: In C, `memcpy` is used to copy the content of a block of memory from one location to another. The result is undefined, though, if the source and destination blocks overlap. LLBMC checks that this does not happen.

Memory Leaks: Memory leaks occur when blocks of memory are allocated, but never de-allocated. For long running programs this might cause an out-of-memory situation. LLBMC checks for memory leaks as described in [26].

User Defined Assertions: In addition to the built-in assertions, LLBMC supports checking user defined properties expressed in C via C's `assert` function or the LLBMC-specific `__llbmc_assert`.[4] Assumptions can also be specified using the built-in function `__llbmc_assume`.

BMC Specific Assertions: Finally, LLBMC is able to automatically detect insufficient bounds for nested function calls and loop iterations that cause BMC to be incomplete since not all programs executions are considered in these cases.

3 LLVM

LLBMC uses the LLVM compiler framework (versions from 2.7 through 3.0) and its intermediate representation LLVM-IR [18]. This makes it possible to use LLBMC on programs that are written in several programming languages, since compiler front-ends for, amongst others, C and C++, are available. The main target of LLBMC is bounded model checking of C programs, but C++ programs that do not use exception handling or run-time type information (RTTI) are also supported.

LLVM's intermediate representation is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers. A program in LLVM-IR consists of type definitions, global variable declarations, and the program itself, which is represented as a set of functions, each consisting of a graph of basic blocks. Each basic block in turn is a list of instructions, where the instruction set can broadly be split into six types:

1. Three-address-code (TAC) instructions working on registers or constants.
2. The memory access instructions `load` and `store`.
3. Address calculations using `getelementptr`.
4. Conditional and unconditional branch instructions, `phi` instructions.
5. Function call instructions.
6. Bit-level instructions like extensions, truncations, and type casts.

[4] `__llbmc_assert` is used only for specification purposes and not checked at runtime.

Here, (conditional and unconditional) branch instructions are only allowed as the last instruction of a basic block. The branch instructions between basic blocks induce a *basic block graph*, in which edges are annotated with the condition under which the transition between the two basic blocks is taken.

Programs in LLVM-IR are in *static single assignment (SSA)* form, i.e., each (scalar) variable is assigned exactly once in the static program. Assignments to scalar variables can thus be treated as logical equivalences. Due to its restricted instruction set, the use of SSA form, and its low-level nature, converting an LLVM-IR program into a logical representation is considerably easier than operating on the source code of a high-level programming language.

The simple C program given in Fig. 1 is used as a running example. This program is converted into the LLVM-IR program also shown in Fig. 1 by the C front-end `llvm-gcc` (on a 32-bit architecture using the optimization level `-O2`).

```
union U {
    char c[4];
    struct { int v: 31; int s: 1; } t;
    int i;
};

void __llbmc_main(char n) {
    union U *u; char *p; int i;
    u = malloc(sizeof(union U));
    p = u->c;
    u->t.s = 1;
    u->t.v = 0;
    p[0] = n;
    __llbmc_assert(u->i == INT_MIN);
}
```

```
define void @__llbmc_main(i8 %n) {
entry:
  %0 = call i8* @malloc(i32 4)           ; u = malloc(sizeof(
  %1 = bitcast i8* %0 to i32*            ;              union U));
  store i32 -2147483648, i32* %1         ; u->t.s = 1; u->t.v = 0;
  store i8 %n, i8* %0                    ; p[0] = n;
  %2 = load i32* %1                      ; u->i
  %3 = icmp eq i32 %2, -2147483648       ;        == INT_MIN ?
  %4 = zext i1 %3 to i32
  call void @__llbmc_assert(i32 %4)
  ret void
}
```

Fig. 1. Example C program. It is converted into the given LLVM-IR program by the C front-end `llvm-gcc`. The function `__llbmc_main` is taken as the starting point for BMC.

Notice that the low-level bit-field and union operations have been replaced by word-level instructions by the front-end.

4 The Approach of LLBMC

The overall approach of LLBMC is as follows: First, an LLVM compiler front-end (such as `clang` or `llvm-gcc`) is used in order to convert a C program into an LLVM-IR program. This LLVM-IR program is then converted into LLBMC's internal logical representation ILR. The ILR formula is simplified by LLBMC using rewrite rules before being passed to an SMT solver. If the SMT solver finds a satisfying assignment (corresponding to a bug in the program), this can be converted into a counterexample, first on the ILR level and then on the LLVM-IR level. The approach is summarized in Fig. 2.

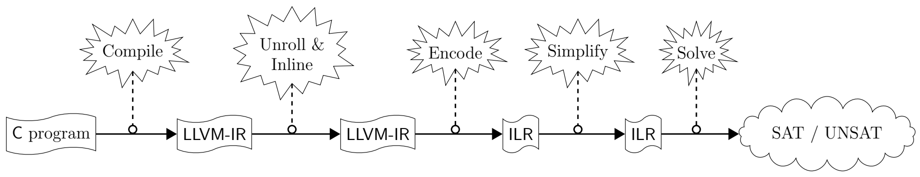


Fig. 2. LLBMC's approach

4.1 From LLVM-IR to ILR

After parsing the LLVM-IR program, a number of transformations are applied to it (e.g., loops are unrolled and functions are inlined a fixed number of times and the control flow graph is simplified).[5] The transformed program is then converted into ILR, which is a representation of a formula in the logic of bit-vectors and arrays with some extensions that, e.g., handle the special semantics of memory allocation instructions like `malloc` and `free`. This format closely follows LLVM's instruction set, but differs from LLVM-IR in that it provides an explicit state object for the memory content as well as for the state of the memory allocation system. These state objects encode the dependencies between memory access instructions and `malloc`/`free`, respectively. With an explicit representation of the memory state, dependencies between memory-related instructions in LLVM (which were implicitly given by the ordering of the operations) are made explicit in the ILR formula. This makes the expressions in ILR order-independent.

[5] LLBMC accepts arbitrary LLVM-IR programs as input and does not depend on any optimizations performed by the compiler. For efficiency reasons, LLBMC internally runs LLVM's `mem2reg` pass in order to promote stack memory to registers when possible. Furthermore, the `indvars` pass is used in order to automatically determine the (static) number of loop iterations for certain kinds of simple loops.

Translation of LLVM's three-address-code, memory access, address calculation, and bit-level instructions is straightforward, since these instructions are part of the theory of bit-vectors and arrays—or can easily be encoded into it.

`phi` instructions are a common tool in compiler IRs that use SSA form. They are used to select the correct value for a variable from a set of previous values (e.g., when control flow merges after an *if-then-else* statement). In general, a `phi` expression in ILR has the form

$$i' = \texttt{phi}\ [i_1, c_1] \ldots [i_n, c_n]$$

where the value that the variable i' takes is one of $i_1, \ldots, i_n$, depending on which of the conditions $c_1, \ldots, c_n$ is true. The conditions c_j are mutually exclusive and cover all possible cases, i.e., the value of i' is always uniquely determined.

For SMT solvers, a `phi` expression can be translated into a sequence of ITE (if-then-else) operators (written in C syntax below):

$$i' = c_1 \ ?\ i_1 : (c_2 \ ?\ i_2 : (\ldots(c_{n-1} \ ?\ i_{n-1} : i_n)\ldots))$$

The conditions c_j are not given explicitly on the LLVM-IR level, though. Instead, basic blocks are used as designators. These basic blocks refer to the immediate predecessor in the basic block graph from which the current basic block has been reached. It thus becomes necessary to compute the conditions c_j. This is accomplished as follows. An *execution condition* $c_{\mathsf{exec}}(b)$ is associated with each basic block b. Execution conditions can be calculated recursively. Let $P(b)$ denote the set of predecessors of b in the basic block graph, and let $t(b, b')$ be the condition under which the transition from basic block b to b' is taken (the edge label in the basic block graph). Then

$$c_{\mathsf{exec}}(b) = \bigvee_{b' \in P(b)} \big(c_{\mathsf{exec}}(b') \wedge t(b', b)\big)$$

if $P(b) \neq \emptyset$, and $c_{\mathsf{exec}}(b) = \top$ otherwise. Then, the basic block b' in a `phi` instruction on the LLVM-IR level that occurs in the basic block b can be replaced by the condition $c_{\mathsf{exec}}(b') \wedge t(b', b)$ on the ILR level.

Notice that each $c_{\mathsf{exec}}(b)$ requires only linear space in the number of predecessors of the basic block b if the recursive definition is not expanded but encoded by introducing new Boolean variables for each $c_{\mathsf{exec}}(b)$ and $t(b, b')$ instead.

4.2 Adding Checks to the ILR Formula

After the initial ILR formula has been generated, it is annotated with `LLBMC`'s built-in checks. Most of these checks are supported by a predicate that is part of ILR, e.g., there are `no_overflow`, `valid_access`, and `valid_free` predicates. Then, an instruction that can possibly overflow is guarded by an assertion that no overflow occurs, a memory access instruction is guarded by an assertion that the access is valid, and so on.

After converting the LLVM-IR program from Fig. 1 into ILR and adding the predicates for the built-in checks, the ILR formula shown in Fig. 3 is obtained. Here, assertions are encoded in such a way that only the first error in the program is reported.

```
i8 %n = nondef()
i8* %0 = nondef()
heap %1 = malloc(%initialHeap, %0, i32_4)
bool %2 = valid_malloc(%initialHeap, %0, i32_4)
assert(%2, "valid_malloc")
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32_2147483648)
bool %5 = valid_access(%1, %3, i32_4)
bool %6 = and(%2, %5)
bool %7 = not(%2)
bool %8 = or(%7, %6)
assert(%8, "valid_store")
mem %9 = store(%4, %0, %n)
bool %10 = valid_access(%1, %0, i32_1)
bool %11 = and(%6, %10)
bool %12 = not(%6)
bool %13 = or(%12, %11)
assert(%13, "valid_store")
i32 %14 = load(%9, %3)
bool %15 = valid_access(%1, %3, i32_4)
bool %16 = and(%11, %15)
bool %17 = not(%11)
bool %18 = or(%17, %16)
assert(valid_load, %18)
bool %19 = compare(EQ, %14, i32_2147483648)
bool %20 = and(%16, %19)
bool %21 = not(%16)
bool %22 = or(%21, %20)
assert(%22, "custom")
```

Fig. 3. ILR formula obtained for the LLVM-IR program from Fig. 1

4.3 Simplification of the ILR Formula

Similar to [25], LLBMC uses term rewriting in order to simplify the ILR formula before passing it to an SMT solver (LLBMC uses Boolector [4] by default, but also supports STP [13] and Z3 [22]). Most of the rewrite rules used by LLBMC are rather simple and correspond to constant propagation or simple arithmetical and logical properties. In total, approximately 150 (conditional) rewrite rules have been implemented in LLBMC in order to simplify the ILR formula.

Before the ILR formula is passed to the SMT solver, ILR's predicates for built-in checks are expanded if they are not already supported by the SMT solver:

- Arithmetic overflow detection is supported by many current SMT solvers. Otherwise, it can be encoded in bit-vector logic directly (see, e.g., [5]).
- Checks for logic and arithmetic shift exceeding the bit-width can easily be encoded in bit-vector logic using suitable comparison expressions. The same is true for invalid memory allocations (i.e., memory allocations that are "too big") and overlapping memory regions in memcpy.

– Invalid memory access, invalid `free`, and memory leak detection is more complex. Their encoding is discussed in detail in Sect. 5.

After expanding the predicates for the built-in checks and rewrite-based simplifications of the formula from Fig. 3, the formula shown in Fig. 4 is obtained.

```
i8  %n = nondef()
i8* %0 = nondef()
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32__2147483648)
mem %9 = store(%4, %0, %n)
i32  %14 = load(%9, %3)
bool %19 = compare(EQ, %14, i32__2147483648)
assert(%19, "custom")
```

Fig. 4. ILR formula obtained by simplifying the ILR formula from Fig. 3

4.4 Counterexample Generation

The simplified ILR formula is then passed to an SMT solver for the logic of bit-vectors and arrays. If the formula is satisfiable, any satisfying assignment corresponds to a bug in the program. By mapping ILR variables to the corresponding instructions in the LLVM-IR program and simulating execution with these values, a trace of the LLVM-IR program that exhibits the bug can be obtained. The bug exhibited by assigning -128 to `n` (and where `malloc` returns the address `0x7ffffffc`) in the running example is displayed in Fig. 5.

5 Encoding Memory Checks

In this section it is described how the memory-related check predicates are expanded into formulas that can be handled by current SMT solvers. The following discussion only considers the heap. Memory blocks on the heap are allocated using `malloc` and de-allocated using `free`. In ILR, these functions take the form

$$h' = \texttt{malloc}(h, p, s)$$
$$h' = \texttt{free}(h, p)$$

where h, h' are (explicit but abstract) heap allocation states, p is a pointer, and s is the size (in bytes) of the memory block that is to be allocated by `malloc`. Notice that `malloc` takes the pointer p as a parameter and does not provide it as a return value. In the conversion from LLVM-IR to ILR, `malloc` is always preceded by a new pointer variable declaration for p, and `malloc` intuitively adds suitable constraints on this pointer. The heap allocation state h' returned by `malloc` can then be considered as having these constraints added. The `free` function modifies the heap allocation state in such a way that the (currently allocated) memory block starting at address p is de-allocated.

```
define void @__llbmc_main(i8 %n) {      ; i8 %n = -128
entry:                                  ; executed
  %0 = call i8* @malloc(i32 4)          ; 0x7ffffffc
  %1 = bitcast i8* %0 to i32*           ; 0x7ffffffc
  store i32 −2147483648, i32* %1
    ; [0x7ffffffc] -> [0x00 0x00 0x00 0x80]
  store i8 %n, i8* %0
    ; [0x7ffffffc] -> [0x80 0x00 0x00 0x80]
  %2 = load i32* %1                     ; -2147483520
  %3 = icmp eq i32 %2, −2147483648      ; 0
  %4 = zext i1 %3 to i32                ; 0
  call void @__llbmc_assert(i32 %4)     ; FAILED
}
```

Fig. 5. Error trace exhibiting a bug in the running example

LLBMC supports two different encodings for the memory checks: a "global" encoding (following [26]) and a "local" encoding (following [12]). In the global encoding, the memory check predicates are expanded by taking the whole formula into consideration at once. In contrast, the local approach is based on conditional rewrite rules that only take the immediate arguments of the predicates into account. As an example, the expansion of the valid-access predicate is discussed below, the remaining memory-related check predicates are handled similarly, see [26,12] for details.

The valid-access predicate has the form

$$\texttt{valid-access}(h, p, s)$$

where h is a heap allocation state, p is a pointer, and s is the size (in bytes) of the memory block that is to be accessed. The intended semantics of this predicate is that it is true in exactly those cases where the memory region $[p, p+s)$ is contained within a memory block that is currently allocated in h.

The "global" encoding of valid-access is given below. The encoding of valid-access(h, p, s) iterates over all mallocs that potentially took place when obtaining the heap allocation state h. valid-access(h, p, s) is then true if a malloc that actually took place allocated a memory block that contains $[p, p+s)$ and if this memory block was not de-allocated since then.

$$\texttt{valid-access}(h, p, s) \quad \equiv$$
$$\bigvee_{\substack{h' \preceq h \\ I:\, h' = \texttt{malloc}(h'', q, t)}} \Big(c_{\mathsf{exec}}(I) \;\wedge\; q \leq p \;\wedge\; p + s \leq q + t \;\wedge\; \neg\texttt{deallocated}(h', h, q) \Big)$$

$$\texttt{deallocated}(h, h', p) \quad \equiv \quad \bigvee_{\substack{h \preceq h^* \preceq h' \\ I:\, h^* = \texttt{free}(h'', q)}} \Big(c_{\mathsf{exec}}(I) \;\wedge\; p = q \Big)$$

Here, $c_{\mathsf{exec}}(I)$ is the execution condition of (the basic block containing the) instruction I. $h' \preceq h$ means that h' is a (direct or indirect) predecessor of h in the history of heap allocation states.

The "local" encoding of `valid-access` is given in the following. It uses conditional rewrite rules of the form $C \mid \ell \longrightarrow r$, expressing that ℓ can be rewritten to r if the condition C can be evaluated to true. A memory access in the "empty" heap allocation state ε is never valid (first rewrite rule). An access within an allocated memory block is always valid (second rewrite rule), and, e.g., an access that partially overlaps with a memory block that is getting de-allocated is never valid (last rewrite rule).

$$\begin{aligned}
\texttt{contained}(p,s,q,t) &:= p \le q \ \wedge\ q+t \le p+s \\
\texttt{disjoint}(p,s,q,t) &:= p+s \le q \ \vee\ q+t \le p
\end{aligned}$$

$$\begin{aligned}
& & \texttt{valid-access}(\varepsilon,p,s) \\
& & \longrightarrow\ \bot \\
\texttt{contained}(p,s,q,t) &\mid & \texttt{valid-access}(\texttt{malloc}(h,p,s),q,t) \\
& & \longrightarrow\ \top \\
\neg\texttt{contained}(p,s,q,t) &\mid & \texttt{valid-access}(\texttt{malloc}(h,p,s),q,t) \\
& & \longrightarrow\ \texttt{valid-access}(h,q,t) \\
\neg\texttt{valid-free}(h,p) &\mid & \texttt{valid-access}(\texttt{free}(h,p),q,t) \\
& & \longrightarrow\ \texttt{valid-access}(h,q,t) \\
\texttt{valid-free}(h,p) \wedge \texttt{disjoint}(p,\texttt{bsize}(h,p),q,t) &\mid & \texttt{valid-access}(\texttt{free}(h,p),q,t) \\
& & \longrightarrow\ \texttt{valid-access}(h,q,t) \\
\texttt{valid-free}(h,p) \wedge \neg\texttt{disjoint}(p,\texttt{bsize}(h,p),q,t) &\mid & \texttt{valid-access}(\texttt{free}(h,p),q,t) \\
& & \longrightarrow\ \bot
\end{aligned}$$

Here, `valid-free` determines whether a `free` is valid, i.e., whether it changes the heap allocation state. `bsize` determines the size of the (currently allocated) memory block beginning at p. See [12] for details on their encodings.

6 Evaluation

In order to evaluate `LLBMC`'s performance, we compared it with two other BMC tools: the C Bounded Model Checker `CBMC` [8] and the Efficient SMT-Based Context-Bounded Model Checker `ESBMC` 1.16 [10].[6] `CBMC` 3.9 contains significant changes concerning the memory model since the typed memory model has been replaced by a (mostly) byte-oriented model [17]. Since this new memory model is less mature than `CBMC`'s typed memory model, we also included the previous version of `CBMC` (3.8) in the comparison.

Benchmarks for the comparison were selected from a variety of papers and sources in order to minimize any kind of bias. In total, 175 C programs were included. The benchmark selection did not include any C++ programs since

[6] `F-Soft` [15] and `SMT-CBMC` [1] are not publicly available.

ESBMC does not support C++ and CBMC's C++ support is still very rudimentary. We have, however, successfully used LLBMC on 57 C++ programs containing advanced features such as multiple inheritance, STL containers, and templates.[7] All four examples presented in [21] and all benchmarks mentioned in [1] were included in the evaluation. All of the benchmarks from the NEC Laboratories America benchmark suite[8] were considered, but only those without infinite loops were chosen (otherwise BMC is incomplete for all loop unrolling depths). A family of four benchmarks implementing queues was constructed by us. Eight examples provided in SLAyer's web interface[9] were included, as well as all examples distributed with the Static Modular Assertion ChecKer SMACK[10] [24], except for those where non-trivial loop invariants were used (which are not supported by either of the evaluated tools). Ten examples from the URBiVA distribution [20] were added as well. Finally, two sets of worst-case execution time benchmark suites were added to the selection of benchmarks: the SNU[11] and the WCET[12] [14] suites. The complete benchmark collection and LLBMC itself are available at http://baldur.iti.kit.edu/llbmc/.

In order to compensate for different default settings, CBMC was run with the options --bounds-check, --div-by-zero-check, --pointer-check, and --overflow-check and ESBMC was run with the option --overflow-check. Otherwise, CBMC and ESBMC where run with their default settings, in particular concerning the choice of SAT or SMT solvers. For LLBMC, the C programs were converted to LLVM-IR using llvm-gcc (version 2.8) with all compiler optimizations switched off. Furthermore, LLBMC was configured to use Boolector as its SMT solver. The loop unrolling and function inlining bounds were set to the lowest possible values to detect a bug or show that no bug is present.

The evaluation was performed on an Intel® Core™ 2 Duo machine with 2.4GHz running Ubuntu Linux 11.04. For each benchmark, the memory limit was set to 2.5GB and the time limit was set to 15 minutes. The results of the comparison are shown in Table 1.

Notice that LLBMC is able to successfully solve (i.e., find bugs in) over 18% more benchmarks than the best other tool in the comparison. The evaluation, however, contains two incorrect results reported by LLBMC:

- In the benchmark family WCET: in the benchmark containing Duff's device (duff.c), a loop is not recognized as such by LLVM. Because of this, LLBMC incorrectly reports insufficient loop unrolling bounds.
- In the benchmark family NECLA: LLVM's optimizations (even using the compiler setting -O0) cause information about signedness of an arithmetic

[7] For these C++ programs, CBMC 3.8 correctly solves nine programs, fails to handle 41 programs, and produces incorrect results for seven programs. For CBMC 3.9, the numbers are four, 53, and zero, respectively.

[8] http://www.nec-labs.com/research/system/systems_SAV-website/

[9] http://rise4fun.com/SLAyer

[10] http://www.zvonimir.info/projects/

[11] http://www.cprover.org/satabs/examples/SNU_Real_Time_Benchmarks/

[12] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Table 1. Results of the evaluation. "N" denotes the number of instances in a benchmark family. "S" denotes the number of successfully solved instances (correctly detected bugs or absence of bugs proved), "O" the number of times the tool ran out of time or memory, "F" the number of failures to handle the input program, and "I" the number of incorrect results (i.e., the tool reports a non-existing "bug" or misses a bug).

Benchmark		LLBMC				CBMC 3.8				CBMC 3.9				ESBMC 1.16			
Family	N	S	O	F	I	S	O	F	I	S	O	F	I	S	O	F	I
[21]	4	**4**	0	0	0	**4**	0	0	0	2	0	2	0	1	1	2	0
[1]	32	**32**	0	0	0	28	4	0	0	28	4	0	0	31	0	1	0
NECLA	45	**44**	0	0	1	34	4	5	2	29	2	9	5	31	4	6	4
Queue	4	**4**	0	0	0	3	1	0	0	3	1	0	0	3	1	0	0
SLAyer	8	**8**	0	0	0	5	0	0	3	4	0	0	4	4	0	1	3
SMACK	38	**38**	0	0	0	30	3	0	5	16	0	0	22	31	0	0	7
SNU	6	**6**	0	0	0	5	0	1	0	5	0	1	0	**6**	0	0	0
URBiVA	10	**10**	0	0	0	9	0	0	1	5	0	0	5	4	1	5	0
WCET	28	26	1	0	1	**27**	1	0	0	**27**	1	0	0	26	1	0	1
Total	175	**172**	1	0	2	145	13	6	11	119	8	12	36	137	8	15	15
%		**98.3**	0.6	0.0	1.1	82.9	7.4	3.4	6.3	68.0	4.6	6.9	20.6	78.3	4.6	8.6	8.6

operation to be lost. LLBMC then does not check the operation for signed arithmetic overflow and misses an overflow bug.

Notice that both CBMC and ESBMC have a significantly larger number of incorrect results, i.e., report more non-existing "bugs" or miss bugs.

The cactus plot in Fig. 6 compares the run-times of the four tools. Benchmarks that could not be handled or where an incorrect result was reported are considered as time-outs. The plot clearly shows that LLBMC produces more correct results in a shorter amount of time than any of the competing tools. Also notice the decrease in the number of correct results between CBMC 3.8 and 3.9.

7 Related Work

Bounded model checking of hardware was introduced by Biere *et al.* in 1999 [3] as an alternative to symbolic model checking using binary decision diagrams (BDDs) [6]. In 2004 Clarke *et al.* were the first to describe the application of BMC to software (more specifically, C programs) [8].

Also in 2004, NEC Laboratories America implemented a bounded model checking approach for C programs in the tool F-Soft as described in [15]. They differentiate their tool from CBMC mainly through a basic block-based approach instead of an SSA-based approach. Several static program analysis techniques are performed on the control-flow graph in order to simplify the BMC problem.

In 2009, Armando *et al.* extended CBMC to use SMT solvers instead of encoding the problem directly into SAT [1]. Results from that paper clearly show the benefits of using SMT solvers w.r.t. formula size and execution time compared to a direct SAT encoding as done by CBMC and F-Soft.

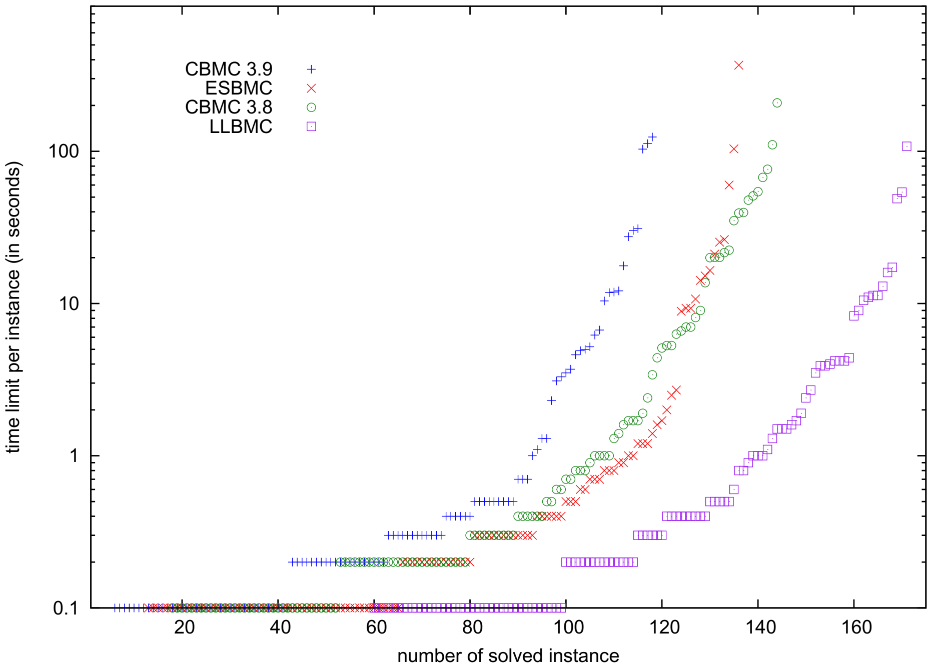


Fig. 6. Cactus plot comparing LLBMC, CBMC 3.8, CBMC 3.9, and ESBMC

Recently, Cordeiro *et al.* presented ESBMC [10], which is based on CBMC but uses an SMT solver instead of a SAT solver. The main novelty of ESBMC is its added support for bug finding in multi-threaded software.

Milicevic and Kugler introduced an approach for model checking of software based on SMT and the theory of lists [21]. While that approach avoids the boundedness limitation of BMC, the evaluation in [21] indicates that it does not scale comparably to BMC based approaches.

Symbolic execution is a different approach to bug detection in programs. In contrast to BMC, which encodes all paths up to a bounded length in a single formula, symbolic execution performs a symbolic path exploration that considers the paths separately. The constraints obtained for each path are solved using SAT or SMT solvers. Recent symbolic execution tools include KLEE [7] for C programs and KLOVER [19], which extends KLEE for C++ programs. Both KLEE and KLOVER perform symbolic execution on the level of LLVM-IR.

A recent tool that combines features of symbolic execution and BMC is LAV [27]. Like KLEE, KLOVER, and LLBMC, the tool LAV also operates on the level of LLVM-IR programs.

Out of the numerous static checking programs, at least Calysto [2] and SMACK [24] operate on the level of LLVM-IR as well.

For related work concerning memory models, we refer to [26,12].

8 Conclusions and Future Work

This paper has presented LLBMC, a tool for bounded model checking of C/C++ programs. LLBMC uses the LLVM compiler framework to translate C/C++ programs into LLVM's intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. An empirical evaluation on a large collection of C programs has shown that LLBMC compares favorably to CBMC [8] and ESBMC [10], both in run-time and in number of found bugs. Furthermore, LLBMC has successfully been used on over 50 non-trivial C++ programs containing advanced features such as multiple inheritance, STL containers, and templates, making it (to the best of our knowledge) the first BMC tool that can handle non-trivial C++ programs.

For future work, we are currently working on lifting the error trace of the LLVM-IR program to an error trace of the C program by using debug information generated by the compiler front-end. We are also planning to determine (and iteratively adapt) loop unrolling and function inlining bounds automatically: starting with low bounds for function inlining and loop unrolling, they are gradually increased based on the results of previous runs of LLBMC. Since the C++ support in LLBMC is currently preliminary and incomplete, we are planning to extend the support for BMC of C++ programs. Similar to [19], support for exception handling and run-time type information (RTTI) needs to be added to LLBMC. Finally, LLBMC could be extended in the direction of software verification (as opposed to bug finding) using k-induction, similar to how this was recently done for CBMC [11].

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. STTT 11(1), 69–83 (2009)
2. Babić, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: Proc. ICSE 2008, pp. 211–220 (2008)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
5. Brummayer, R.D.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Johannes Kepler Universität, Linz, Austria (2009)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. IC 98(2), 142–170 (1992)
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI 2008, pp. 209–224 (2008)
8. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

9. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. ENTCS 254, 85–103 (2009)
10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proc. ASE 2009, pp. 137–148 (2009)
11. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software Verification using k-Induction. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011)
12. Falke, S., Merz, F., Sinz, C.: A theory of C-style memory allocation. In: Proc. SMT 2011, pp. 71–80 (2011)
13. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
14. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks – past, present and future. In: Proc. WCET 2010, pp. 137–147 (2010)
15. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. TCS 404(3), 256–274 (2008)
16. Kim, M., Kim, Y., Kim, H.: Unit testing of flash memory device driver through a SAT-based model checker. In: Proc. ASE 2008, 198–207 (2008)
17. Kröning, D.: CBMC release 3.9 announcement on (December 19, 2010), cprovergooglegroups.com
18. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO 2004, pp. 75–88 (2004)
19. Li, G., Ghosh, I., Rajan, S.: KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 609–615. Springer, Heidelberg (2011)
20. Maric, F., Janicic, P.: URBiVA: Uniform Reduction to Bit-Vector Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 346–352. Springer, Heidelberg (2010)
21. Milicevic, A., Kugler, H.: Model Checking using SMT and Theory of Lists. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 282–297. Springer, Heidelberg (2011)
22. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
23. Post, H., Sinz, C., Küchlin, W.: Towards automatic software model checking of thousands of Linux modules—A case study with Avinux. STVR 19(2), 155–172 (2009)
24. Rakamarić, Z., Hu, A.J.: A Scalable Memory Model for Low-Level Code. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 290–304. Springer, Heidelberg (2009)
25. Sinha, N.: Symbolic program analysis using term rewriting and generalization. In: Proc. FMCAD 2008, pp. 1–9 (2008)
26. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Proc. SSV 2010 (2010)
27. Vujosevic-Janicic, M., Kuncak, V.: Development and Evaluation of LAV: an SMT-Based Error Finding Platform. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSSTE 2012. LNCS, vol. 7152, pp. 98–113. Springer, Heidelberg (2012)

The Marriage of Exploration and Deduction

Rupak Majumdar

Max-Planck Institute for Software Systems, Germany
rupak@mpi-sws.org

State space exploration based on abstraction and refinement has been the cornerstone of several successful software verification tools from the last decade. While these tools have made impressive progress in verifying control-dominant properties of code, most prominently in the domain of device drivers, their applications to more data-intensive properties have been limited. In particular, we focus on *parameterized systems*, which define infinite families of systems, one for each value of the parameter. Many real-life software systems, for example, memory management units or cache coherence protocols can be modeled as parameterized systems (parameterized, e.g., by the number of processes and memory locations), We want to perform *uniform verification* of parameterized systems, where we show a formula is an invariant of every member in the parameterized family.

The primary verification strategy for parameterized systems is *deduction*. For safety verification, the user guesses an inductive invariant for the parameterized family, and uses decision procedures to check that the guess is indeed correct. The deductive approach has the advantage of better scalability (through reducing the problem to a few SMT queries), expressiveness (invariants are usually quantified, and modern decision procedures have good quantifier instantiation heuristics), and avoids unbounded refinement loops that plague abstraction-refinement based model checkers. However, it depends on the ingenuity of the user to provide appropriate inductive invariants.

We shall show how combinations of deductive and explorative techniques can lead to automatic verification for parameterized systems, or at least, reduce the manual effort required to come up with appropriate inductive invariants. We demonstrate the applicability of the technique through two case studies: the verification of transactional memories and of cache coherence protocols. We also outline some challenges in applying the methods, and directions which require further research.

Modeling and Validating the Train Fare Calculation and Adjustment System Using VDM++

Nguyen Van Tang[1,⋆], Daisuke Souma[1], Goro Hatayama[2], and Hitoshi Ohsaki[1]

[1] Research Team for Verification and Specification
National Institute of Advanced Industrial Science and Technology, Japan
[2] Core Technology Development Department
Omron Social Solutions Co., Ltd, Japan
t.nguyen@aist.go.jp

Abstract. The Train Fare Calculation and Adjustment System (TFCAS), developed by the OMRON Corporation, is a large-scale and complex system that helps passengers buy tickets and adjust their train fare on the railways across Japan. In this paper we present the results and experiences gained in a collaborative research project between AIST and OMRON, in which VDM++ has been applied to formalize TFCAS's specifications and validate its consistency as well as reliability properties. An executable VDM++ model can be used to raise the level of the quality of the informal system specification, the efficiency of existing system test-suites, and the quality of real implementation. The application of VDM++ enables us to detect 32 erroneous issues in the original informal specification document. Moreover, we also show how the development process can be improved in a front-loading manner using the formal method VDM++.

1 Introduction

More and more cities are using train (subway) systems for public transportation in daily life. Among other important systems on a train system, such as, sensors, the interlocking system, and the train control system, *etc.*, the train fare calculation system can be seen as the nerve center system of the train services. This system serves as the key communication between the passengers, the station workers, and even the train companies. As a consequence, train fare calculation systems are becoming useful, but these systems are complex and large-scale systems, and they include many complicated data structures and algorithms for computing routes (lists of stations, and a station is a pair of line name and station name), the fares of routes, and fare adjustments, *etc.* These systems are embedded into the ticket vending and fare adjustment machines at every station and are used daily. For example, the Paris train system has about 550 stations, the London train system 750 stations, the New York train system

⋆ Corresponding author.

1000 stations, and the Kansai train system (Osaka and its suburb) has 1300 stations. The Kanto area train system (Tokyo and its suburbs) includes about 3000 train stations, and approximately 40 million people use the train fare calculation system everyday[1]. There are many kinds of tickets in Japan, *e.g.*, regular tickets, section tickets, tickets for children, etc. Once a passenger buys a ticket, it is worth noting that if a ticket price is higher than the fare of the trip *because of a bug or error*, the passenger loses money. Otherwise, if a ticket price is lower than the fare of the trip, the company loses money. Because there are so many customers, a tiny error on the ticket price could turn out to be a huge loss of money for the railways company or passengers. Furthermore, once bugs are found, the Japanese government will ask the company to make reports and fix the bugs. This kind of task is expensive and time-consuming, and will affect the prestige of the company. Due to these social factors, there is a crucial need to reduce risks in the development process of train service systems.

The Train Fare Calculation and Adjustment System (TFCAS for abbreviation), promoted by Japanese railways companies, is a complex system for train fare calculation of the railways across Japan. The train fare calculation system currently in use was developed over 10 years ago, and some modules (components) of this system can be reused. The system was embedded into ticket vending machines and fare adjustment machines at every station in Japan. The task of OMRON is to develop a new system with more useful features by reusing legacy modules of the old system. The new system includes new algorithms and user-friendly interfaces in order to help users find the shortest route or the best fare between two stations, and to help users perform fare adjustment (or pay a penalty) if *they buy the wrong ticket for the trip or there are errors with the system, etc.* With such convenient and useful services, the system is necessarily a complex and large-scale system with a huge database. Our goal is to find an efficient way to improve the specification of TFCAS and guarantee that the design of the new system is consistent and compatible with the legacy modules.

Formal software development has been extensively studied (for instance, see [1,4] for surveys) in recent years. VDM++, in particular, is an object-oriented extension of VDM-SL [9], which is a formal specification language. VDM++ is supported by an industry-strength tool set VDMTools[2] owned and developed by CSK Systems and extended from the former IFAD VDM-SL Toolbox [9]. The tools offer syntax checking, type checking and proof obligation generation capabilities, code generators for C++ or Java, a mechanism for a VDM-C++ connection, an attractive printer, a CORBA-based Application Programming Interface (API) and links to external tools for UML modeling to support round-trip engineering. VDMTools also supports Japanese, so that transforming the informal specifications (in Japanese) to the formal model in VDM++ is relatively easy.

In this paper we present the results and experiences gained from a case study where VDM++ has been applied to formalize and verify reliability properties of the TFCAS specifications. This work has been performed in a joint project

[1] `http://en.wikipedia.org/wiki/Tokyo_subway`
[2] `http://www.vdmtools.jp/en/`

of the OMRON Corporation and the National Institute of Advanced Industrial Science and Technology (AIST). Our approach is as follows: Because the system reuses several legacy components of the existing system, we first employ a VDM-C++ connection to integrate the newly added/maintained components (written in VDM++) with the legacy ones (written in C/C++) to check for consistency and compatibility. Second, we perform model-based testing on the integrated model against system requirements. Third, we apply conformance testing to check whether the practical implementation of the system (concurrently developed in C/C++ by OMRON) conform with the VDM++ model. With this overall goal, the main results of this paper are as follows:

– First, the application we present here demonstrates the benefits derived from establishing an explicit formal model of the system. During the formal specification of the TFCAS system, *31 erroneous issues* have been found in the informal specification documents of the system. The errors are categorized into several groups: ambiguities, omissions, duplications, description of specifications, and inconsistencies.
– Second, besides the expected improvement of the informal specification documents, the efficiency of the system test environment (using cluster machines) to test the functionality of the system has been analyzed and then increased. We present the model-based and conformance testing of several properties with different characteristics (types of tickets, companies, etc) of the fare adjustment functionality. In the first stage of testing, we use cluster machines to perform testing for about 850000 test cases. With this test setting, redundant test cases as well as wrong-data test cases are detected. Also, *one subtle defect* of system design was found in the testing phase. This helps us not only to test for correctness but also to optimize the set of test cases by providing test designers. In order to have more coverage, OMRON engineers will apply our approach to test more fare adjustment functions against about 10 million test cases in the next coming months.
– Third, the engineers at OMRON wanted to know how such a model would improve informal specifications and support maintenance. An existing change request has been chosen in order to investigate the role of an explicit model in the needed modification process, such as the openness of the system. With a formal model in VDM++, whenever new features are added into the system, we only need to formalize the newly added/maintained parts in the VDM++ specification. In this way, it is more convenient to combine legacy components with the newly added components to test for consistency and compatibility.

The remainder of this paper is organized as follows. Section 2 describes an overview of the project including goals and our approach, as well as an overview of the system. In Section 3 we present our method of using VDM++ for formalizing and validating the TFCAS specifications. We also report certain identified issues in this section too. Section 4 describes our strategy and the process of testing and validating the system. Section 5 reports on our intensive experiments of testing and some types of defects. Section 6 presents several lessons learned from our project. In Section 7 we discuss related works. Finally, Section 8 concludes the paper and outlines future work.

2 Project Overview

2.1 The TFCAS System: An Informal Overview

The system is embedded into ticket vending and fare adjustment machines at every station in Japan. The main modules and notions of this system are split up as follows:

– **Ticket Vending Machines:** A train system may be owned by one or several companies. Each company may have several lines, e.g., the company A has the $L1$ and $L2$ lines. Station here is an abstract notion. A station is not only a location name, but a pair of line names and location names. This is because there are many lines owned by different railway companies in Japan. A route is simply a list of stations. There are many types of tickets and fares, *e.g.*, the fare for routes, the fares for children, the fares in a company, special fares between two companies, *etc.* The first module of the system is embedded in ticket vending machines at every station in Japan. With a user-friendly interface, passengers can easily query the shortest trip (or a trip with the cheapest fare) between two arbitrary stations using the system, so that the passengers can select and buy tickets using cash or cards.

– **Fare Adjustment Machines:** The second important module of the system consists of fare adjustment functionality. This part is embedded into fare adjustment machines inside every station in Japan. In the USA and most European countries, fare adjustment is not allowed. In Japan, fare adjustment is allowed when passengers arrive at the destination station by using the *fare adjustment* machines. For example, a passenger buys a ticket for a trip from station A to station B. He/she however can use this ticket to take another trip, say from station A to station C, provided that he/she has to make a fare adjustment to get out of station C. Although the idea is simple, those required fare adjustment functionalities are complicated in practice. Because there are about 20 types of tickets, fare adjustment depends on the types of tickets and the relationship between companies as well as many other conditions, *e.g.,* conditions for children or disabled persons.

2.2 Goals and Approach of the Project

In this project we aim at the following goals: *producing consistent and precise specifications, enhancing the quality of the design documents, improving development processes, extensive testing at different levels, checking consistency between the reused components and the newly developed components.*

We have decided to use the formal specification language VDM++ [5] with VDMTools [5] since they support the description and validation of large-scale

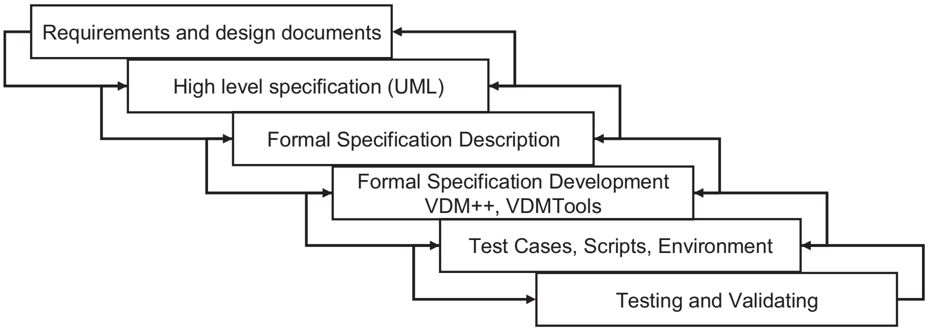

Requirements and design documents
High level specification (UML)
Formal Specification Description
Formal Specification Development
VDM++, VDMTools
Test Cases, Scripts, Environment
Testing and Validating

Fig. 1. An overview of our approach

systems. Readers are referred to [5,6] for more details of the VDM++ technology. Our approach consists of the following main steps (also graphically depicted as the waterfall model in Figure 1):

- **High Level Specification:** To formalize the specification, our approach first exploits the Unified Modeling Language (UML) to obtain a class diagram of the system. This graphical view of the system makes it easier to understand an overview of classes, data types, abstract methods and functions, and relationships among classes.
- **Formal Specification Description:** In this step, we define abstract data types and classes, as well as implicit specifications of functions using pre/post conditions. Syntax and type checking were performed in this step to check consistencies of data types.
- **Formal Specification Development:** To get a comprehensive and detailed specification, we refine the specification step-by-step by defining the detailed contents of functions.
- **Test Environment Development:** After finishing the functionality specification, we establish data and the environment for testing the system. We import the test data from the OMRON corporation. The test tickets were given in XML format, therefore, we have to translate test data from XML to VDM format.
- **Testing and Validating:** We perform integrated and conformance testing on the system. The test results in VDM were compared to the test results of the C/C++ system (concurrently implemented by OMRON). If there are any differences found from the comparison, we report the problem to the company to check more details of the formal specification and the real implementation.
- **Training Industrial Users:** Finally, a part of results has been opened in public in the form of a workshop attended by (employees of) the Japanese railway companies.

3 VDM++ Model of the TFCAS

In this section the techniques and results of modeling the system in VDM++ are presented. First, an abstract UML model of the object-oriented architecture is outlined. Second, the specification of *several classes and functions* serves to demonstrate the specification language of VDM++. The section concludes with statistics of the errors found in the natural language specifications by the modeling phase.

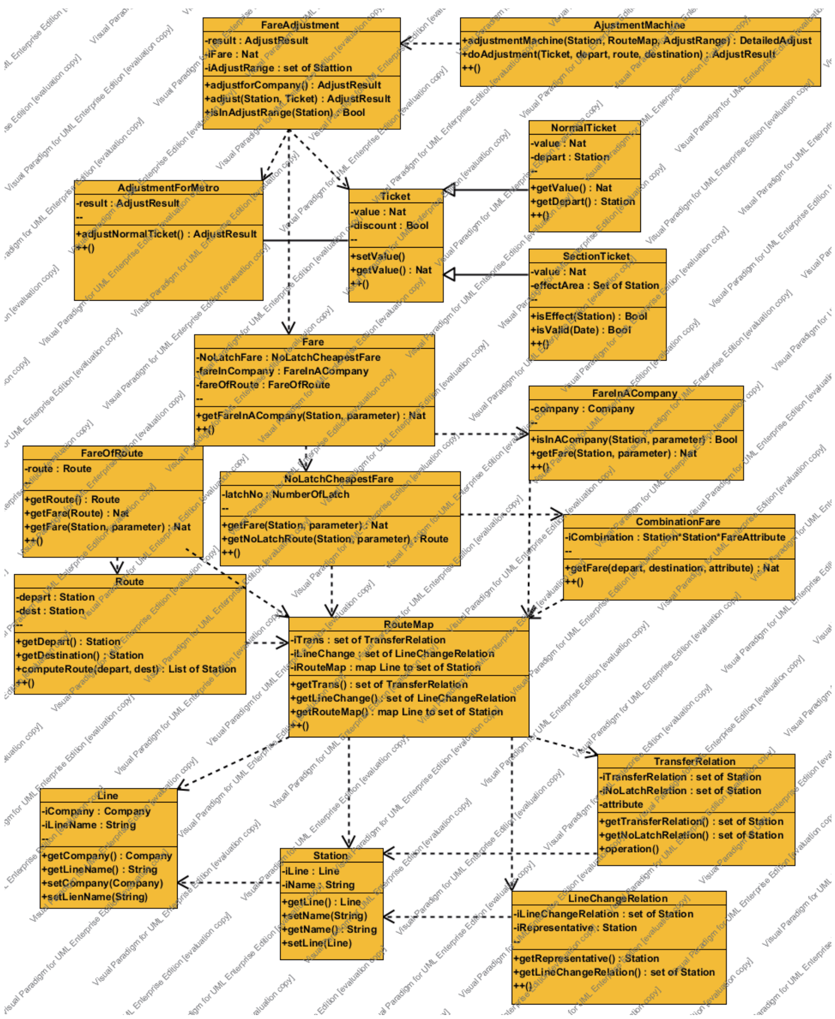

FareAdjustment
-result : AdjustResult
-iFare : Nat
-iAdjustRange : set of Stattion
+adjustforCompany() : AdjustResult
+adjust(Station, Ticket) : AdjustResult
+isInAdjustRange(Station) : Bool
AjustmentMachine
+adjustmentMachine(Station, RouteMap, AdjustRange) : DetailedAdjust
+doAdjustment(Ticket, depart, route, destination) : AdjustResult
++()
NormalTicket
-value : Nat
-depart : Station
+getValue() : Nat
+getDepart() : Station
++()
AdjustmentForMetro
-result : AdjustResult
+adjustNormalTicket() : AdjustResult
++()
Ticket
-value : Nat
-discount : Bool
+setValue()
+getValue() : Nat
++()
SectionTicket
-value : Nat
-effectArea : Set of Station
+isEffect(Station) : Bool
+isValid(Date) : Bool
++()
Fare
-NoLatchFare : NoLatchCheapestFare
-fareInCompany : FareInACompany
-fareOfRoute : FareOfRoute
+getFareInACompany(Station, parameter) : Nat
++()
FareInACompany
-company : Company
+isInACompany(Station, parameter) : Bool
+getFare(Station, parameter) : Nat
++()
FareOfRoute
-route : Route
+getRoute() : Route
+getFare(Route) : Nat
+getFare(Station, parameter) : Nat
++()
NoLatchCheapestFare
-latchNo : NumberOfLatch
+getFare(Station, parameter) : Nat
+getNoLatchRoute(Station, parameter) : Route
++()
CombinationFare
-iCombination : Station*Station*FareAttribute
+getFare(depart, destination, attribute) : Nat
++()
Route
-depart : Station
-dest : Station
+getDepart() : Station
+getDestination() : Station
+computeRoute(depart, dest) : List of Station
++()
RouteMap
-iTrans : set of TransferRelation
-iLineChange : set of LineChangeRelation
-iRouteMap : map Line to set of Station
+getTrans() : set of TransferRelation
+getLineChange() : set of LineChangeRelation
+getRouteMap() : map Line to set of Station
++()
TransferRelation
-iTransferRelation : set of Station
-iNoLatchRelation : set of Station
-attribute
+getTransferRelation() : set of Station
+getNoLatchRelation() : set of Station
+operation()
Line
-iCompany : Company
-iLineName : String
+getCompany() : Company
+getLineName() : String
+setCompany(Company)
+setLienName(String)
Station
-iLine : Line
-iName : String
+getLine() : Line
+setName(String)
+getName() : String
+setLine(Line)
LineChangeRelation
-iLineChangeRelation : set of Station
-iRepresentative : Station
+getRepresentative() : Station
+getLineChangeRelation() : set of Station
++()

Fig. 2. An Abstract UML Class Diagram of the System

3.1 Architecture of the System

In order to describe the specification graphically, an object-oriented architecture reflecting the physical components has been chosen. Figure 2 shows the main classes of the system and the relationship between these classes in UML notation. We give abstract information of TCFAS in the UML class diagram. In the following, we illustrate the architecture of the TFCAS system in VDM++ notation. In this paper, we only give abstract definitions of several classes such as *Line*, *Station*, *RouteMap*, and *NormalTicketAdjustment*.

```
class Line
  types
    public Company = token;

  instance variables
    iCompany  : Company;
    iLineName : seq1 of char;

  operations
    public Line : Company * seq1 of char ==> Line
      Line(aCompany,aLineName) ==(
        iCompany  := aCompany;
        iLineName := aLineName;
    );

    public getCompany : () ==> Company
      getCompany() == return iCompany
    ;

    public getLineName : () ==> seq1 of char
      getLineName() == return iLineName
    ;
end Line
```

In the definition of the class *Line*, we use an abstract data type *token* for *Company*. Later, for testing purposes, each element of this type can be constructed by using $mk\_token(``ACompanyName")$, e.g., $mk\_token(``X")$ indicates the company (with name) X. Once we have the class *Line*, the class *Station* can be defined in VDM++ as follows:

```
class Station
  instance variables
    iStationName : seq1 of char;
    iCompany     : Line`Company;
  operations
    public Station : Line`Company * seq1 of char  ==> Station
      Station(aCompany,aStationName) ==(
        iCompany     := aCompany;
        iStationName := aStationName;
      )
    ;
    public getName : () ==> seq1 of char
      getName() == return iStationName
    ;
    public getCompany : () ==> Line`Company
      getCompany() == return iCompany
    ;
end Station
```

The *RouteMap* class contains information about the line-change and transfer relation as well as the mapping from a set of lines to lists of stations. The abstract of this class is defined as follows. The additional *invariant constraints* (inv) means that the company of all stations (in the range of iRouteMap) coincides with and the company of the domain of iRouteMap.

```
class RouteMap

instance variables
  iRouteMap      : map Line to seq1 of Station;
  iLineChangeSet : set of LineChangeRelation;
  iTransferSet   : set of TransferRelation;

  inv
    forall xLine in set dom iRouteMap
    &
      forall xStation in set elems iRouteMap(xLine)
      &
        xStation.getCompany() = xLine.getCompany()
  ...
end  RouteMap
```

3.2 A Simple Example of the TFCAS's Functionality

Within this project, fare adjustments are critical functions of train fare calculation systems. Therefore, in this paper, we focus mainly on the functions of fare adjustments.

In the original specification, there are many functions for fare adjustments. The content of each function depends on not only the type of the ticket, but also the destination station. In what follows, we present the simplest case of fare adjustment function for regular tickets. The fare adjustment for other types of tickets are much more complicated. For presentation purposes, we omit their details in this paper.

```
class NormalTicketAdjustment
types
  public <extra_fare>     = nat;
  public AdjustResult = <Unnecessary> | <Impossible> | <Certificate>
                        | <Extra_fare>;
operations

  protected adjustFor1Company : NormalTicket * Station * Line`Company
                                       ==>           AdjustResult
      adjustFor1Company(aTicket,adjustStation,aCompany) ==
      let fareOfTrip = cheapestFareInACompany(aTicket.getDeparture(),
                  adjustStation,aCompany,aTicket.getFareAttribute())
      in
      if (fareOfTrip <= aTicket.getValue())
      then return <unnecessary>
      else return fareOfTrip - aTicket.getValue()
  pre
      aTicket.TypeCode() in set {<standard>}
  ;
  ...
end NormalTicketAdjustment
```

Each company may have different policies for fare adjustment if the departure stations and the stations they get out at are in the same company. In other words, the fare adjustments mainly depend on policies of companies and the relationships among companies (in the cases where there are more than 2 companies involved). In the above example of fare adjustment for regular tickets, the operation `adjustFor1Company` describes the simple case of fare adjustment for the single company. More precisely, the specification of this function in VDM++ can be interpreted as follows:

1. `<Extra_fare>` is a type of natural numbers this type represents an extra fare that a client needs to pay.
2. `<AdjustResult>` is an enumerated type of VDM++ to represents four possible cases of fare adjustments: `<Unneceassry>` indicates that it is not necessary to adjust the fare. `<Impossible>` indicates that it is impossible to adjust the fare. `<Certificate>` indicated that passengers can adjust fares, and receive a certificate that helps the them to get out of the station without paying more money. `<Extra_fare>` indicates that passengers can adjust fares by paying an extra fare to get out of the station.

3. The operation `adjustFor1Company`: If the value of the ticket is greater than the fare of the trip, the passenger does not need to adjust fares. Otherwise, the passenger needs to adjust fare by paying an extra fee (fine): *fareOfTrip - aTicket.getValue()*.

3.3 Analysis of Identified Issues

During the formal specification of the TFCAS system, a total number of 31 erroneous issues were found in the original informal specification document. It is significant to note that most of the errors were detected in the modeling (specification) process. The errors in the design documents related to specifications detected in the modeling phase are shown in Table 1.

Table 1. Statistics of Errors before the Integration Test

Reasons of errors	number of errors
Ambiguities	8
Omission	6
Duplication	3
Description of specification	5
Inconsistencies	9

4 Testing and Validating Process

Testing always needs some reference to test against in order to decide whether the behavior of the system is correct or not. In our project, we apply the following test strategy. First, we test the VDM++ model of the system to check whether the specification satisfies system requirements. Second, we use the VDM-C++ connection functionality to integrate the newly added components (written in VDM++) with the legacy modules of the old system (written in C/C++). We then perform testing on the resulting integrated system to check whether the newly added components are consistent with the legacy ones. Third, the implementation of software in practice, however, may not conform with the specification (model). We therefore perform *conformance testing* to check whether the implementation in OMRON conforms to the specification. Our testing and validating process is described in Figure 3. The main phases of the process can be described as follows:

– **Phase 1. Modeling in VDM++:** From the original design document of the system, we formalize the design specification of the system in VDM++.
– **Phase 2. Implementation in C/C++:** To keep the development schedule on time, concurrent with the modeling phase in VDM++, a OMRON developing team starts implementing the system in C/C++. Both teams (research and development) frequently communicate with each other to get information and feedback to improve the modeling and implementation step by step.

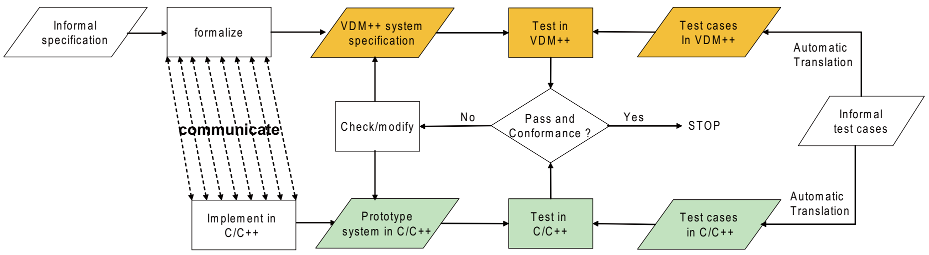


Fig. 3. The Modeling and Model-based/Conformance Testing Process

- **Phase 3. Test cases preparation:** The informal test cases were created and stored in XML format by engineers and domain experts in OMRON and Japan Railways Companies. In the current state, all test cases are for fare adjustment of regular tickets, section tickets, and combination tickets. We imported the test data from OMRON corporation. The test tickets were given in XML format, therefore, we have to write a program to automatically translate test data from XML to VDM++ and C/C++ formats, respectively.
- **Phase 4. Testing in VDM++:** After modeling the new design specification in VDM++, we connect the components in VDM++ with the legacy modules (e.g., functions to compute the fare between two stations of the same company) in C/C++. We then test the VDM++ model of the system for the test cases obtained from Phase 3.
- **Phase 5. Testing in C/C++:** Independently, the development team at OMRON performs testing for the *implementation* (in C/C++) of the system with the same set of test cases obtained from Phase 3.
- **Phase 6. Conformance Testing and Refinement:** We analyze and make comparisons between the test results of the system requirements (i.e., expected results), the system design (i.e., test results of the VDM++ model), and the real implementation (i.e., test results of the implementation). In particular, there are several possible cases as follows: First, if all tests from both teams pass, both the VDM++ model and its implementation validate the required properties. Second, if some tests for the VDM++ model fail, there are errors in the design or specification steps. We need to recheck and revise the design document or the VDM++ model to make all tests pass. Third, if some tests for the implementation fail, it means that there are mistakes in the implementation (i.e., the implementation does not conform with the specification in VDM++). We need to inspect and revise the implementation to make all tests pass.

5 Experiments

The total duration of the project was 24 months including the training of industrial users. There were 6 team members in this project, and the average age was about 30 years old. We have implemented a part of the specification (in Japanese) of the TFCAS system in VDM++. One person is mainly responsible for VDM++ programming the others are responsible for checking the specification in VDM++ and the original documents, the test environment preparation, the VDM-C++ connection, and the testing performance. However, all members of the project frequently discuss with each other the details of the specification. Two members joined the project in the last six months (of the entire 24 months duration) to learn VDM++ and to transform our technique to OMRON. The specification is implemented by CSK VDMToolbox 8.2.0 on Windows XP. The average productivity for the formal specifications was about 400 lines of VDM++ code per engineer per month (approximately 160 hours), including the time used for examination of requirements, and for testing the formal specification. In summary, the details are given in Table 2.

Table 2. Parameters, Cost Estimation and Testing Information

Contents	**Size**
a part of an informal document	6400 lines (161 A4 pages)
functional specification in VDM++	9400 lines
data specification in VDM++	800000 lines
C++ code to connect VDM++ with C++ legacy modules	1000 lines
number of test cases	839771
research team	6 persons
cost estimation of the research project	6 man-year
Test cases	**Time**
395,000 regular tickets	1:16:00
275,000 section tickets	6:28:00
170,000 pair of normal and section tickets	41:42:00

We tested the functions of the fare adjustment functionalities against a large number of test cases. In particular, the set of test cases is categorized into three groups: *fare adjustment for regular tickets, fare adjustment for section tickets, and fare adjustment for combination tickets (pairs of normal and section tickets).* We perform testing in parallel using cluster machines (Intel Xeon X7350 2.93GHz Quad Core A-16 of 1TB memory). Concurrently, the team of OMRON has developed a test environment in C/C++ and performed testing with the same set of test cases as ours. The running time of testing in C/C++ is 10 times faster than that of in VDM++. This is because VDM++ is a (interpreter) specification language, not a programming language. However, because we use cluster machines and do testing in parallel, we think that the testing time of VDM++ is acceptable.

Thanks to VDM++, the test results helps us in detecting some problems of the specification of functions for the fare adjustment as follows:

1. Some tested results are found to be different from the expected ones. Based on those test cases, we detected and fixed 1 subtle error related to functions of fare adjustment for combination tickets in the informal specification document of the systems.
2. Testing in VDM++ enables us to detect the redundant test cases supplied by OMRON. Also, a set of test cases that could not be processed by VDM++ was discovered. We realized that this set of test cases have the wrong data, i.e., the data is out of the range of the specification. Altogether, such detected information helps test designers to optimize the set of test cases.
3. Based on a comparison of the tested results of the model (in VDM++) and that of the implementation (in C/C++), differences were detected. We inspected both the codes of VDM++ and C/C++ and found that some functions in C/C++ were incorrectly implemented. This led us to early discussion with the team at OMRON, which allowed us to fix errors to make C/C++ implementation conform with the model.

6 Lessons Learned

In this section we discuss several interesting positive and negative observations, which could be used in exploitation of the methodology in other industrial settings.

– **Tests and Reviews of Specifications:** First, with a VDM++ model of the system, it is easier for us to check consistency and correctness of the system specification. As reported in the previous section, the formal method contributes to enhancing the quality of deliverables at the design phase of the development process, because the formal specification can be syntax-checked, type-checked and tested. In our experience, achieving the same level of quality with natural language and the UML specifications that are subjected to more limited checking and inspection is difficult.
– **Conformance Testing and Refinement:** Second, we analyze the test results and make a comparison among system requirements (i.e., expected results), system design (i.e., test results of VDM++ model), and real implementation (i.e., test results of implementation). In particular, there are several cases as follows: (1) if all tests from both teams pass, both the VDM++ model and its implementation validate the required properties; (2) if some tests for the VDM++ model fail, it means that there are errors in the design or specification steps. We need to recheck and revise the design document or the VDM++ model to make all tests pass; (3) if some tests for the implementation fail, it means that there are mistakes in implementation (i.e., the implementation does not conform with the specification in VDM++). We need to inspect and revise the implementation to make all tests pass.
– **Cost Estimation and Comparison:** Third, from the practical point of view, we found that using the VDM++ formal method can help us to reduce the development cost for the TFCAS system. Based on experiences obtained

from previous projects, a comparison is made between the cost using the formal method VDM++ and the estimated cost using the standard approach (natural language and UML) for software development. At the beginning of the process, the cost is higher for using the formal method for specification compared to the cost for the standard approach. The reason is that, at this step, we need more engineers to make a team for formal specification as well as the expense of buying formal method tools (i.e., VDM++ Toolbox), etc. However, using the formal method, errors and mistakes can be detected earlier in the design phase, which helps us to reduce costs in testing and maintenance afterwards. In contrast to this, the standard approach cannot detect errors early, and thus errors still remain until the integration and system testing phase. It is very costly to detect errors and modify the system. As a result, the total cost of the formal method approach is smaller than that of the standard one (see Figure 4 for details).

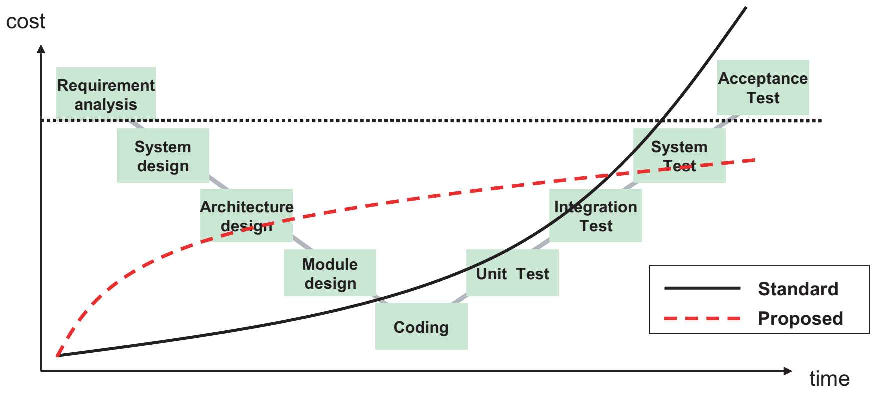


Fig. 4. Cost Comparison Diagram

7 Related Work

The successful transfer of formal method technology into industrial practice has been a goal of researchers and practitioners for several decades. Many attempts apply formal specification and validation into industrial projects have been reported in recent surveys [1,4]. Many attempts have been made to apply formal methods to railways and their associated systems such as sensors, interlocking systems and train control systems [2,7,10]. Eriksson has applied formal methods to the problem of verifying interlocking systems with great success for over ten years in this, notably on behalf of Banverket (the Swedish National Rail Administration) [10]. This approach works by creating two mathematical models: the first is that of the interlocking system and consists of rules, and the second is of the topological aspects of the railway yard for which the interlocking system has been designed. Verification proceeds by proving that a signalling principle holds for the interlocking model in the topology model of the railyard. NP-Tools software produced by the company Prover[3] have been used for the verification.

In [2], Chiappini et al. presented a method for formalization and validation of a subset of the *European Train Control System* (ETCS). In their approach, they first exploit a subset of the Unified Modeling Language (UML) and a fragment of the Property Specification Language (PSL). The constraint language mixes Linear time temporal logic (LTL), regular expressions, first-order logic and hybrid aspects related to real-time evolution. Second, they created a new specification language so-called *Control Natural Language* (CNL) by combining the constraint language and a small fragment of English expressions. Next, the specification and requirements of the ETCS system was formalized by CNL. Finally, CNL constraints were verified by an extended version of the NuSMV model checker, which is able to deal with continuous variables.

Successful industry use has been a major driver behind the development of methods and tools for VDM in the past fifteen years, influencing the development of tool support [5]. The ConForm studied at British Aerospace [3] strongly suggested that benefits could be gained from early-stage abstract modeling using a tool-supported formalism, even with only testing available as an analytic tool. Further, VDM++ was used to demonstrate how maintenance (of a *Voice Communication System used in Air Traffic Control*) is supported by a formal model at FREQUENTIS [7]. More recent applications have used the object-oriented extensions, notably the key components of the TradeOne back-office system developed by CSK systems for the Japanese stock exchange [5] and the development of the Mobile Felica operating system for an integrated circuit for cellular telephone applications by Sony [8]. A common factor in many VDM applications has been the use of formal models as a way of gaining rapid early feedback on requirements and designs. The goal of providing such useful feedback during the development of embedded systems has motivated our work reported here.

8 Conclusions

We have reported a case-study using VDM++, a formal specification method and tool, to formalize and validate the Train Fare Calculation and Adjustment System (TFCAS). The validation technique allows us to check consistency and correctness of the specification with respect to the system requirements. A total number of 32 erroneous issues have been found in the original informal specification document. The application of VDM++ was viewed as highly successful in not only keeping our project on schedule but also in enhancing the quality of deliverables at the design phase of the development process. Thanks to the formal method VDM++, the results of our project were positively evaluated by domain experts and potential users external to the Japanese railway consortium.

We conclude that the formal method VDM++ is suitable for reaching high quality by continuously (and simultaneously) improving the formalization and

[3] `http://www.prover.com/`

development process step by step, working closely together with team members. For future work, we plan to explore more efficient and comprehensive ways to combine VDM++ and a theorem prover/model checker/SAT solver for formal specification and verification. We hope to use such a combined approach instead of only testing to verify more complicated properties.

Acknowledgments. We would like to express our great appreciation to the anonymous reviewers and editors for giving us valuable and helpful comments to improve the paper.

References

1. Bicarregui, J., Fitzgerald, J.S., Larsen, P.G., Woodcock, J.: Industrial Practice in Formal Methods: A Review. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 810–813. Springer, Heidelberg (2009)
2. Chiappini, A., Cimatti, A., Macchi, L., Rebollo, O., Roveri, M., Susi, A., Tonetta, S., Vittorini, B.: Formalization and validation of a subset of the European Train Control System. In: Proc. of ICSE 2010, pp. 109–118. ACM Press (2010)
3. Fitzgerald, J.S., Brookes, T.M., Green, M.A., Larsen, P.G.: Formal and Informal Specifications of a Secure System Component: First Results in a Comparative Study. In: Naftalin, M., Bertrán, M., Denvir, T. (eds.) FME 1994. LNCS, vol. 873, pp. 35–44. Springer, Heidelberg (1994)
4. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)
5. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object oriented Systems. Springer, New York (2005)
6. Larsen, P.G., Fitzgerald, J.S., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems Using VDM. Int. J. Software and Informatics 3(2-3), 305–341 (2009)
7. Hörl, J., Aichernig, B.K.: Validating Voice Communication Requirements Using Lightweight Formal Methods. IEEE Software 17(3) (2000)
8. Kurita, T., Nakatsugawa, Y.: The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip. Int. J. Software and Informatics 3(2-3), 343–355 (2009)
9. The VDM Tool Group, IFAD. User Manual for the IFAD VDM++ Toolbox. The Institute of Applied Computer Science, Forskerparken 10, 5230 Odense M, Denmark/Europe, 1.0 edn. (September 1997) Doc.Id.: IFAD-VDM-50
10. Eriksson, L.: Formal Verification of Railway Interlockings. Swedish National Rail Administration Technical Report 4 (1997)

Formalized Verification of Snapshotable Trees: Separation and Sharing

Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft

IT University of Copenhagen
{hame,fisi,birkedal,sestoft}@itu.dk

Abstract. We use separation logic to specify and verify a Java program that implements snapshotable search trees, fully formalizing the specification and verification in the Coq proof assistant. We achieve *local* and *modular* reasoning about a tree and its snapshots and their iterators, although the implementation involves shared mutable heap data structures with no separation or ownership relation between the various data.

The paper also introduces a series of four increasingly sophisticated implementations and verifies the first one. The others are included as future work and as a set of challenge problems for full functional specification and verification, whether by separation logic or by other formalisms.

1 Introduction

This paper presents a family of realistic but compact challenge case studies for modular software verification. We fully specified and verified the first case study in Coq, using a domain-specific separation logic [10] and building upon our higher-order separation logic [2]. As future work we plan to verify the other implementations with the presented abstract interface specification. We believe this is the first mechanical formalization of this approach to modular reasoning about implementations that use shared heap data with no separation or ownership relation between the various data.

The family of case studies consists of a single interface specification for snapshotable trees, and four different implementations. A *snapshotable tree* is an ordered binary tree that represents a set of items and supports taking readonly *snapshots* of the set, in constant time, at the expense of slightly slower subsequent updates to the tree. A snapshotable tree also supports iteration (enumeration) over its items as do, e.g., the Java collection classes. The four implementations of the snapshotable tree interface all involve shared heap data as well as increasingly subtle uses of destructive heap update.

For practical purposes it is important that the same interface specification can support verification of multiple implementations with varying degrees of internal sharing and destructive update. Moreover, the specification must accommodate any number of data structure (tree) instances, each having any number of iterators and snapshots, each of which in turn can have any number of iterators. Most importantly, we show how we can have local reasoning (a frame rule) even though the tree and its snapshots share mutable heap data.

We welcome other solutions to the specification and verification of this case study; indeed R. Leino has already made one (unpublished) using Dafny [11].

The Java source code of the case studies of all four implementations and the Coq source is available at `http://www.itu.dk/people/hame/snapshots.tgz`.

Section 2 presents the interface of the case study data structure, shows an example use, and outlines four implementations. Section 3 gives a formal specification of the interface using separation logic and verifies the example code. Sections 4 and 5 verify the first implementation.

2 Case Study: Snapshotable Trees

The case study is a simplified version of snapshotable treesets from the C5 collection library [8].

2.1 Interface: Operations on Snapshotable Trees

Conceptually, a snapshot of a treeset is a readonly copy of the treeset. Subsequent updates to the tree do not affect any of its snapshots, so one can update the tree while iterating over a snapshot. Taking a snapshot must be a constant time operation, but subsequent updates to the tree may be slower after a snapshot has been taken. Implementations (Section 2.3) typically achieve this by making the tree and its snapshots share parts of their representation, gradually unsharing it as the tree gets updated, in a manner somewhat analogous to copy-on-write memory management schemes in operating systems.

All tree and snapshot implementations implement the same ITree interface:

```
public interface ITree extends Iterable<Integer> {
  public boolean contains(int x);
  public boolean add(int x);
  public ITree snapshot();
  public Iterator<Integer> iterator();
}
```

These operations have the following effect:

- `tree.contains(x)` returns true if the item is in the tree, otherwise false.
- `tree.add(x)` adds the item to the tree and returns true if the item was not already in the tree; otherwise does nothing and returns false.
- `tree.snapshot()` returns a readonly snapshot of the given tree. Updates to the given tree will not affect the snapshot. A snapshot cannot be made from a snapshot.
- `tree.iterator()` returns an iterator (also called enumerator, or stream) of the tree's items. Any number of iterators on a tree or snapshot may exist at the same time. Modifying a tree will invalidate all iterators on that tree (but not on its snapshots), so that the next operation on such an iterator will throw ConcurrentModificationException.

We include the somewhat complicated `iterator()` operation because it makes the distinction between a tree and its snapshots completely clear: While it is illegal to modify a tree while iterating over it, it is perfectly legal to modify the tree while iterating over one of its snapshots. Also, this poses an additional verification challenge when considering implementations with rebalancing (cases A2B1 and A2B2 in Section 2.3) because `tree.add(item)` may rebalance the tree in the middle of an iteration over a snapshot of the tree, and that should be legal and not affect the iteration.

Note that for simplicity, items are here taken to be integers; using techniques from [20] it is straightforward to extend our formal specification and verification to handle a generic version of snapshotable trees.

2.2 Example Client Code

To show what can be done with snapshots and iterators (and not without), consider this piece of client code. It creates a treeset `t`, adds three items to it, creates a snapshot `s` of the tree, and then iterates over the snapshot's three items while adding new items (6 and 9) to the tree:

```
ITree t = new Tree();
t.add(2); t.add(1); t.add(3);
ITree s = t.snapshot();
Iterator<Integer> it = s.iterator();
boolean lc = it.hasNext();
while (lc) {
  int x = it.next();
  t.add(x * 3);
  lc = it.hasNext();
}
```

2.3 Implementations of Snapshotable Trees

One may consider four implementations of treesets, spanned by two orthogonal implementation features. First, the tree may be unbalanced (A1) or it may be actively rebalanced (A2) to keep depth $O(\log n)$. Second, snapshots may be kept persistent, that is, unaffected by tree updates, either by path copy persistence (B1) or by node copy persistence (B2):

	Without rebalancing	With rebalancing
Path copy persistence	A1B1	A2B1
Node copy persistence	A1B2	A2B2

The implementation closest to that of the C5 library [8, section 13.10] is A2B2, which is still somewhat simplified: only integer items, no comparer argument, no update events, and so on. In this paper we formalize and verify only implementation A1B1; the verification of the more sophisticated implementations A1B2, A2B1 and A2B2 will be addressed in future work.

Nevertheless, for completeness and in the hope that others may consider this verification challenge, we briefly discuss all four implementations and the expected verification challenges here.

With *path copy persistence* (cases AxB1), adding an item to a tree will duplicate the path from the root to the added node, if this is necessary to avoid modifying any snapshot of the tree. Thus an update will create $O(d)$ new nodes where d is the depth of the tree.

With *node copy persistence* (cases AxB2), each tree node has a spare child reference. The first update to a node uses this spare reference, does not copy the node and does not update its parent; the node remains shared between the tree and its snapshots. Only the second update to a node copies it and updates its parent. Thus an update does not replicate the entire path to the tree root; the number of new nodes per update is amortized $O(1)$. See Driscoll [6] or [8].

To implement ordered trees without rebalancing (cases A1By), we use a Node class containing an item (here an integer) and left and right children; `null` is used to indicate the absence of a child. A tree or snapshot contains a stamp (indicating the "time" of the most recent update) and a reference to the root Node object; `null` if the tree is empty.

To implement rebalancing of trees (cases A2By), we use left-leaning red-black trees (LLRB) which encode 2-3 trees [1,19], instead of general red-black trees [7] as in the C5 library. This reduces the number of rebalancing cases.

To implement iterators on a tree or snapshot we use a class TreeIterator that holds a reference to the underlying tree, a stamp (the creation "time" of the iterator) and a stack of nodes. The stamp is used to detect subsequent updates to the underlying tree, which will invalidate the iterator. Since snapshots cannot be updated, their iterators are never invalidated. The iterator's stack holds its current state: for each node in the stack, the node's own item and all items in the right subtree have yet to be output by the iterator.

Case A1B1 = no rebalancing, path copy persistence In this implementation there is shared data between a tree and its snapshots, but the shared data is not being mutated because the entire path from the root to an added node gets replicated. Hence no node reachable from the root of a snapshot, or from nodes in its iterators' stacks, can be affected by an update to the live tree; therefore no operation on a snapshot can be affected by operations on the live tree. Although this case is therefore the simplest case, it already contains many challenges in finding a suitable specification for trees, snapshots and iterators, and in proving the stack-based iterator implementation correct.

Case A2B1 = rebalancing, path copy persistence In this case there is potential mutation of shared data, because the rebalancing rotations seem to be able to affect nodes just off the fresh unshared path from a newly added node to the root. This could adversely affect an iterator of a snapshot because a reference from the iterator's node stack might have its right child updated (by a rotation), thus wrongly outputting the items of its right subtree twice or not at all. However, this does not happen because the receiver of a rotation (to be moved down) is always a fresh node (we're in case B1 = path copy persistence) and moreover we consider only `add` operations (not `remove`), so the child being rotated (moved up) is also a fresh node and thus not on the stack of any iterator – the rebalancing was caused by this child being "too deep" in the tree. Hence if we were to support `remove` as well, then perhaps the implementation of rotations needs to be refined.

Case A1B2 = no rebalancing, node copy persistence In this case, there is mutation of shared data not observable by the client. For example, a left-child update to a tree node that is also part of a snapshot will move the snapshot's left-child value to the node's extra reference field, and destructively update the left child as required for the live tree. There should be no observable change to the snapshot, despite the change to the data representing it. The basic reason for correctness is that any snapshot involving an updated node will use the extra reference and hence not see the update; this is true for nodes reachable from the root of a snapshot as well as for nodes reachable from the stack of an iterator. When we need to update a node whose extra reference is already in use, we leave the old node alone and create a fresh copy of the node for use in the live tree; again, existing snapshots and their iterators do not see the update.

Case A2B2 = rebalancing, node copy persistence In this case there is mutation of shared data (due both to moving child fields to the extra reference in nodes, and due to rotations), not observable for the client. Since the updates caused by rotations are handled exactly like other updates, the correctness of rebalancing with respect to iterators seems to be more straightforward than in case A2B1.

3 Abstract Specification and Client Code Verification

We use higher-order separation logic [18,3] to specify and verify the snapshotable tree data structure. We build on top of our intuitionistic formalization of HOSL in Coq [2] with semantics for an untyped Java-like language.

To allow implementations to share data between a tree, its snapshots, and iterators and still make it possible for clients to reason locally (to focus only on a single tree / snapshot / iterator), we will use an idea from [10] (see also the verification of Union-Find in [9]). The idea is to introduce an abstract predicate, here named H, global to each tree data structure consisting of a single tree, multiple snapshots, and multiple iterators. This abstract predicate H is parameterized by a finite set of disjoint *abstract structures*. We have three kinds of abstract structures: Tree, Snap, and Iter. The use of H enables a client of our specification to consider each abstract structure to be separate or disjoint from the rest of the abstract structures and thus the client can reason modularly about client code using only those abstract structures she needs; the rest can be framed out. Since the abstract predicate H is existentially quantified, the client has no knowledge of how an implementation defines H (see [3,16] for more on abstract predicates in higher-order separation logic). The implementor of the tree data structure has a global view on the tree with its snapshots and iterators, and is able to define which parts of the abstract structures are shared in the concrete heap. Section 4 defines H for the A1B1 case from Section 2.3.

The Tree abstract structure consists of a handle (reference) to the tree and a model, which is an ordered finite set, containing the elements of the tree. The Snap structure is similar to Tree. The Iter structure consists of a handle to the iterator and a model, which is a list containing the remaining elements for iteration. Because H is tree-global, exactly one Tree structure must be present ("the tree"), while the number of Snap and Iter structures is not constrained.

3.1 Specification of the ITree Interface

We now present the formal abstract specification of the ITree interface informally described in Section 2.1. The specification also contains five axioms, which are useful for a client and obligations to an implementor of the interface. The specification is parametrized over an implementation class C and the above-mentioned predicate H, and each method specification is universally quantified over the model τ, a finite set of integers and a finite set of abstract structures ϕ.

`interface ITree` $\{$

$\{H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)\}$ `contains(x)` $\{\texttt{ret} = \texttt{x} \in \tau \wedge H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)\}$

$\{H(\{\mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi)\}$ `contains(x)` $\{\texttt{ret} = \texttt{x} \in \tau \wedge H(\{\mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi)\}$

$\{H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)\}$ `add(x)` $\{\texttt{ret} = \texttt{x} \notin \tau \wedge H(\{\mathit{Tree}(\texttt{this}, \{\texttt{x}\} \cup \tau)\} \uplus \phi)\}$

$\{H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)\}$ `snapshot()` $\{H(\{\mathit{Snap}(\texttt{ret}, \tau)\} \uplus \{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)\}$

$\{H(\{\mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi)\}$ `iterator()` $\{H(\{\mathit{Iter}(\texttt{ret}, [\tau])\} \uplus \{\mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi) \wedge$
$\texttt{ret} <: \mathit{Iterator}\}$

(a) $H(\{\mathit{Tree}(\texttt{t}, \tau)\} \uplus \phi) \vdash \texttt{t} : C$

(b) $H(\{\mathit{Snap}(\texttt{s}, \tau)\} \uplus \phi) \vdash \texttt{s} : C$

(c) $\tau = \tau' \wedge H(\{\mathit{Tree}(\texttt{t}, \tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(\texttt{t}, \tau')\} \uplus \phi)$

(d) $H(\{\mathit{Snap}(\texttt{s}, \tau)\} \uplus \phi) \vdash H(\phi)$

(e) $H(\{\mathit{Iter}(\texttt{it}, \alpha)\} \uplus \phi) \vdash H(\phi)$

$\}$

These specifications can be read as follows:

- `contains` requires either a Snap or Tree structure (written as separate specifications) for the `this` handle and some set τ. The structure is unmodified in the postcondition, and the return value `ret` is true if the item `x` is in the set τ, otherwise false.
- `add` requires a Tree structure for the `this` handle and some set τ. The postcondition states that the given item `x` is added to the set τ. The return value indicates whether the tree was modified, which is the case if the item was not already present in the set τ.
- `snapshot` requires a Tree structure for the `this` handle and some set τ. The postcondition constructs a Snap structure for the returned handle `ret` and the set τ. So the Tree and the Snap structure contain the same elements.
- `iterator` requires a Snap structure for the `this` handle and some set τ. The postcondition constructs an Iter structure with the return handle and the set τ converted to an ordered list, written $[\tau]$. The returned handle conforms (written $<:$) to the Iterator specification shown in Section 3.2.

The five axioms state that (a) the static type of the tree is the given class C; (b) the static type of a snapshot is C; (c) the model τ of the tree can be replaced by an equal model τ' [1]; and we can forget about snapshots (d) and iterators (e).

In contrast to the description in Section 2.1 we leave iterators over the tree for future work. We could use the ramification operator [10] to express that any iterators over the tree become invalid when the tree is modified.

The abstract separation can be observed, e.g., in the specification of `add`: it only modifies the model of the Tree structure and does not affect the rest of the abstract structures (ϕ is preserved in the postcondition). Hence the client can reason about calls to `add` locally, independently of how many snapshots and iterators there are.

In our Coq formalization we do not have any syntax for interfaces at the specification logic level [2], but represent interfaces using Coq-level definitions. Appendix A contains the formal representations (ITree, Iterator, Stack).

3.2 Iterator Specification

Our iterator specification is also parametrized over a class IC and a predicate H, and each method specification is universally quantified over a list of integers α and a finite set of abstract structures ϕ.

`interface Iterator<Integer> {`
$\{H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\}$ `hasNext()` $\{\texttt{ret} = (|\alpha| \neq 0) \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\}$
$\{H(\{Iter(\texttt{this}, x :: \alpha)\} \uplus \phi)\}$ `next()` $\{\texttt{ret} = x \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\}$
`}`

The specification of the Iterator interface requires an Iter structure with the `this` handle and some list α. The return value of the method `hasNext` captures whether the list α is non-empty. The Iter structure in the postcondition is not modified. The method `next` requires an Iter structure with a non-empty list $(x :: \alpha)$. The list head is returned and the model of the Iter structure is updated to the remainder of the list.

3.3 Client Code Verification

To verify the client code from Section 2.2 we assume we are given a class C such that ITree C H holds for some H and then verify the client code under the precondition $\{H(\{Tree(\texttt{t}, \{\})\})\}$.

Figure 1 gives a step-by-step proof of the client code from Section 2.2, with client code lines to the left and their postconditions to the right.

After inserting some items (line 1) to the tree, the model contains these items, $\{1, 2, 3\}$. In line 2, a snapshot `s` of the tree `t` is created. The invariant H now consists of the Tree structure and a Snap structure containing the same elements. For the client the abstract structures are disjoint, but in an implementation, they will be realized using sharing. Indeed, for the A1B1 implementation, the concrete

[1] This is explicit for technical reasons: in our implementation H is defined inside a monad [2], and the client should not have to discharge obligations inside the monad.

1: `t.add(2);t.add(1);t.add(3);`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\})\}$
2: `ITree s = t.snapshot();`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$
3: `Iterator<Integer> it = s.iterator();`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, [1,2,3])\})\}$
4: `boolean lc = it.hasNext();`	$\{\mathtt{lc} = \mathtt{true} \wedge H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, [1,2,3])\})\}$
5: `while (lc) {`	**invariant:** $\exists \alpha, \beta. \alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (\lvert\beta\rvert \neq 0) \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\}))\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta)\})$
6: `  int x = it.next();`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (\lvert\beta\rvert \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
7: `  t.add(x * 3);`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (\lvert\beta\rvert \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\} \cup \{3\mathtt{x}\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
8: `  lc = it.hasNext();`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (\lvert\beta'\rvert \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z \mid z \in \alpha\} \cup \{3\mathtt{x}\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
9: `}`	$\{H(\{Tree(\mathtt{t}, \{1,2,3,6,9\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$

Fig. 1. Client code verification

heap will be as shown in Figure 2, where all the nodes are shared between the tree and the snapshot.

In line 3 an iterator `it` over the snapshot `s` is created. To apply the call rule of the `iterator` method, only the Snap structure is taken into account, the rest (the Tree structure) is framed out inside of H (via appropriate instantiation of ϕ in the `iterator` specification). The result is that an Iter structure is constructed, whose model contains the same values as the model of the snapshot, but converted to an ordered list. We introduce the loop condition `lc` in line 4, and again use abstract framing to call `hasNext`.

Lines 5–9 contain a while loop with loop condition `lc`. The loop invariant splits the iteration list $[1, 2, 3]$ into the list α containing the elements already iterated over and the list β containing the remainder. The loop variable `lc` is false iff β is the empty list. The invariant H contains the Tree structure whose model is the initial set $\{1, 2, 3\}$ joined with the set of the elements of α, each multiplied by 3. H also contains the Iter and the Snap structures.

We omit detailed explanation of the remaining lines of verification.

Note that in the final postcondition, the client sees two disjoint structures (axiom (e) is used to forget the empty iterator), but in the A1B1 implementation, the concrete heap will involve sharing, as shown in Figure 3. Only the left subtree is shared by the tree and the snapshot; the root and right subtree were unshared by the first call to `add` in the loop.

In summary, we have shown the following theorem, which says that given any H and any classes C and IC satisfying the ITree and Iterator interface

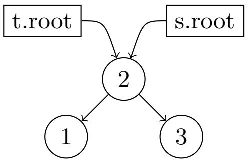


Fig. 2. Heap after snapshot construction

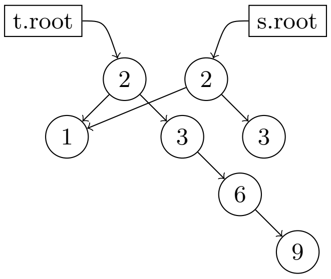


Fig. 3. Live heap after loop

specifications, the client code satisfies its specification. The postcondition states that snapshot `s` contains 1, 2 and 3, and tree `t` contains additionally 6 and 9.

Theorem 1. $\forall H.\forall C.\forall IC.ITree\ C\ H \wedge Iterator\ IC\ H \vdash \{H(\{Tree(\mathtt{t}, \{\})\})\}$ $client\_code\ \{H(\{Tree(\mathtt{t}, \{1, 2, 3, 6, 9\})\} \uplus \{Snap(\mathtt{s}, \{1, 2, 3\})\})\}$

4 Implementation A1B1

In this section we show the partial correctness verification of the A1B1 implementation with respect to the abstract specification from the previous section. This involves defining a concrete H and showing that the methods satisfy the required specifications for this concrete H. The development has been formally verified in Coq (as has the client program verification above).

The Coq formalization uses a shallow embedding of higher-order separation logic, developed for verification of OO programs using interfaces. See [2].

Invariant H is radically different depending on whether snapshots of the tree are present or not. The reason is that method `add` mutates the existing tree if there are no snapshots present, see Section 5 for details. Here we focus on the case where snapshots are present.

The A1B1Tree class stores its data in three fields: the `root` node, a boolean field `isSnapshot`, indicating whether it is a snapshot, and a field `hasSnapshot`, indicating whether it has snapshots. The stamp field mentioned in Section 2.3 is only required for iterators over the tree and so not further discussed here.

The Node class is a nested class of the A1B1Tree with three fields, `item` containing its value, and a handle to the right (`rght`) and left (`left`) subtree.

In the following we use standard separation logic connectives, in particular the separating conjunction $*$ and the points to predicate $\mapsto$.

We now define our concrete H and also the realization of the abstract structures. We first explain the realization of Tree and Snap; the Iter structure is described in Section 4.1. Recall that ϕ ranges over finite sets of abstract structures (Tree, Snap, Iter), with exactly one Tree structure, and recall that H, given a ϕ, returns a separation logic predicate. The definition of H is:

$$H(\phi) \triangleq \exists \sigma. wf(\sigma) \wedge heap(\sigma)\ *\ \sigma \vDash \phi$$

Here σ is a finite map of type ptr $\rightarrow$ ptr $\times \mathbb{Z} \times$ ptr, with ptr being the type of Java pointers (handles), corresponding to the Node class. The map σ must be well-formed (pure predicate $wf(\sigma)$), which simply means that all pointers in the codomain of σ are either `null` or in the domain of σ.

The *heap* function maps σ to a separation logic predicate, which describes the realization of σ as a linked structure in the concrete heap, starting with $\top$:

$$\begin{aligned}\mathit{heap}(\sigma) \triangleq \mathit{fold}\ (\lambda \mathtt{p}\ n\ Q.\ \mathit{match}\ n\ \mathit{with}\ (\mathtt{pl}, v, \mathtt{pr}) \Rightarrow \\ \mathtt{p.left} \mapsto \mathtt{pl} * \mathtt{p.item} \mapsto v * \mathtt{p.rght} \mapsto \mathtt{pr} * Q)\ \top\ \ \sigma\end{aligned}$$

Finally, we present the definition of $\sigma \vDash \phi$ (we defer the definition of $\sigma \vDash \{\mathit{Iter}(\_,\_)\}$ to the following subsection):

$$\begin{aligned}\sigma \vDash \phi \uplus \psi &\triangleq \sigma \vDash \phi\ *\ \sigma \vDash \psi \\ \sigma \vDash \{\mathit{Tree}(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}.\mathit{Node}(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p} * \\ &\qquad \mathtt{ptr.isSnapshot} \mapsto \mathtt{false} * \mathtt{ptr.hasSnapshot} \mapsto \mathtt{true} \\ \sigma \vDash \{\mathit{Snap}(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}.\mathit{Node}(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p} * \\ &\qquad \mathtt{ptr.isSnapshot} \mapsto \mathtt{true} * \mathtt{ptr.hasSnapshot} \mapsto \mathtt{false}\end{aligned}$$

The spatial structure of all the nodes is covered by $\mathit{heap}(\sigma)$ so $\sigma \vDash \phi$ just needs to describe the additional heap taken up by Tree, Snap, and Iter structures.

The pure *Node* predicate is defined inductively on τ below. It is used to express that τ is the finite set of items reachable from `p` in σ.

$$\begin{aligned}\mathit{Node}(\sigma, \mathtt{p}, \tau) \triangleq\ & \big(\mathtt{p} = \mathtt{null} \wedge \tau = \{\}\big) \vee \\ & \big(\mathtt{p} \in \mathit{dom}(\sigma)\ \wedge \exists \mathtt{pl}, v, \mathtt{pr}.\ \sigma[\mathtt{p}] = (\mathtt{pl}, v, \mathtt{pr}) \wedge \\ & \exists \tau_l, \tau_r. \tau = \tau_l \cup \{v\} \cup \tau_r \wedge \\ & (\forall x \in \tau_l. x < v) \wedge (\forall x \in \tau_r. x > v) \wedge \\ & \mathit{Node}(\sigma, \mathtt{pl}, \tau_l) \wedge \mathit{Node}(\sigma, \mathtt{pr}, \tau_r)\big)\end{aligned}$$

The sortedness constraint (a strict total order) in the *Node* predicate enforces implicitly that σ has the right shape: σ cannot contain cycles and the left and right subtrees must be disjoint. The set τ is split into three sets, one with strictly smaller elements (τ_l), the singleton v and with strictly bigger elements (τ_r).

4.1 Iterator

The TreeIterator class implements the Iterator interface. It contains a single field, `context`, which is a stack of Node objects.

The constructor of the TreeIterator pushes all nodes on the leftmost path of the tree onto the stack. The method `next` pops the top node from the stack and returns the value held in that node. Before returning, it pushes the leftmost path of the node's right subtree (if any) onto the stack. The method `hasNext` returns true if and only if the stack is empty.

The verification of the iterator depends on the following specification of a stack class, generic in C. The specification is parametrized over a representation type T and existentially over a representation predicate SR (of type $\mathsf{classname} \rightarrow (\mathsf{val} \rightarrow T \rightarrow \mathsf{HeapAsn}) \rightarrow \mathsf{val} \rightarrow T^* \rightarrow \mathsf{HeapAsn}$). The second argument is the predicate P (of type $\mathsf{val} \to T \to \mathsf{HeapAsn}$) , which holds for every stack element. This specification is kept in the style of [17], although we use a different logic.

```
class Stack<C> {
  ⊤                         new()     SR C P ret nil
  SR C P this α             empty()   ret = (α = nil) ∧ SR C P this α
  SR C P this α ∗ P x t ∧ x : C  push(x)  SR C P this (t :: α)
  SR C P this (t :: α)      pop()     P ret t ∗ SR C P this α
  SR C P this (t :: α)      peek()    P ret t ∗(∀u.P ret u −∗ SR C P this (u :: α))
(a) P v t ⊢ P′ v t ⟹ SR C P v α ⊢ SR C P′ v α
}
```

For the purpose of specifying the iterators over snapshotable trees, we instantiate the type T with $\mathbb{Z}^*$; the model of a node on the stack is a list of integers. Intuitively, this list corresponds to the node value and the element list of its right subtree. The iterator is modelled as a list that is equal to the concatenation of the elements of the stack. We also require that the topmost element of the stack is nonempty (if present). This intuition is formalized in the interpretation of the Iter structure, where SR is a representation predicate of a stack:

$$\sigma \vDash \{Iter(\mathtt{p}, \alpha)\} \triangleq \exists \mathtt{st}.\ \mathtt{p.context} \mapsto \mathtt{st}\ *\ \exists\beta.stack\_inv(\beta, \alpha) \wedge \\ SR\ \mathtt{Node}\ (NS\ \sigma)\ \mathtt{st}\ \beta.$$

To make this definition complete, we provide the definitions of *stack_inv*, which connects the representation of the stack with the representation of the iterator, and the definition of the NS predicate.

$$stack\_inv(xss, ys) \triangleq ys = \mathsf{concat}(xss) \wedge \begin{cases} \top & \text{iff } xss = \mathsf{nil} \\ xs \neq \mathsf{nil} & \text{iff } xss = xs :: xss' \end{cases}$$

$$NS\ \sigma\ \mathtt{node}\ \alpha \triangleq Node(\sigma, \mathtt{node}, \tau) \wedge\ \alpha = [\{x \in \tau | x \geq \mathtt{node.item}\}]$$

These definitions, along with an assumption that SR is the representation predicate of Stack (i.e., fulfills all the method specifications and axioms of *Stack_spec*) suffice to show the correctness of Iter-dependent methods. The axiom present in *Stack_spec* is needed to preserve iterators if some new memory is added to σ: it allows us to replace $(NS\ \sigma)$ with $(NS\ \sigma')$ as a representation predicate of stack objects under certain side conditions.

5 On the Verification of Implemented Code

We now give an intuitive description of how the A1B1 implementation was verified, given the concrete H defined above. We verified the complete implementation in Coq but only discuss the `add` method here. We used Kopitiam [13] to transform the Java code into SimpleJava, the fragment represented in Coq.

Method `add` calls method `addRecursive` with the root node to insert the item into the binary tree, respecting the ordering. Method `addRecursive`, shown below, must handle several cases:

– if there are no snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and destructively inserted into the tree.
– if there are snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and every node on the path from the root to the added node is replicated, so that the snapshots are unimpaired.

The implementation of `addRecursive` walks down the tree until a node with the same value, or a leaf, is reached. It uses the call stack to remember the path in the tree. If a node was added, either the entire path from the root to the added node is duplicated (if snapshots are present) or the handles to the left or right subtree are updated (happens destructively exactly once, the parent of the added node updates its left or right handle, previously pointing to `null`):

```
Node addRecursive (Node node, int item, RefBool updated) {
  Node res = node;
  if (node == null) {
    updated.value = true;
    res = new Node(item);
  } else {
    if (item < node.item) {
      Node newLeft = addRecursive(node.left, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(newLeft, node.item, node.rght);
      else
        node.left = newLeft;
    } else if (node.item < item) {
      Node newRght = addRecursive(node.rght, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(node.left, node.item, newRght);
      else
        node.rght = newRght;
    } //else item == node.item so no update
  }
  return res;
}
```

We now show the pre- and postcondition of `addRecursive` for the two cases where snapshots are present. If the item is already present in the tree, the pre- and postcondition are equal:

$$\begin{gathered}\{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ heap(\sigma) \ * \ wf(\sigma) \wedge Node(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \in \tau\} \\ \texttt{addRecursive(node, item, updated)} \\ \{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ heap(\sigma) \ * \texttt{ret} = \texttt{node}\}\end{gathered}$$

The postcondition in the case that the item is added to the tree extends the map σ to σ', for which the heap layout and the well-formedness condition must hold. The Node predicate uses σ' and the finite set is extended with `item`:

$$\begin{gathered}\{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ \mathit{heap}(\sigma) \ * \ \mathit{wf}(\sigma) \wedge \mathit{Node}(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \notin \tau\} \\ \texttt{addRecursive(node, item, updated)} \\ \{\texttt{updated.value} \mapsto \texttt{true} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ \exists \sigma'.\sigma \subseteq \sigma' \wedge \mathit{heap}(\sigma') \ * \ \mathit{wf}(\sigma') \wedge \mathit{Node}(\sigma', \texttt{ret}, \{\texttt{item}\} \cup \tau)\}\end{gathered}$$

The call to `addRecursive` inside of `add` is verified for each specification of `addRecursive` independently.

To summarize Sections 4 and 5, we state the following theorem, which says there exists an H that given a stack fulfilling the stack specification, the TreeIterator class meets the Iterator specification and the A1B1 class meets the ITree specification, and the constructor for the A1B1Tree establishes the H predicate.

Theorem 2. $\exists H.\mathit{Stack\_spec} \vdash \mathit{Iterator}\ \mathit{TreeIterator}\ H \wedge \mathit{ITree}\ A1B1\ H \wedge \{\top\}\ A1B1\mathit{Tree}()\ \{H(\{\mathit{Tree}(\texttt{ret}, \{\})\})\}$

The client code, independently verified, can be safely linked with the A1B1 implementation!

6 Related Work

Malecha and Morrisett [12] presented a formalization of a Ynot implementation of B-trees with an iterator method. In their case, the iterator and the tree also share data in the concrete heap. However, they can only reason about "single-threaded" uses of trees and iterators: their specification of the iterator method transforms the abstract tree predicate into an abstract iterator predicate, which prohibits calling tree methods until the client turns the iterator back into a tree. In our setup, we have one tree, but multiple snapshots and iterators, and the tree can be updated after an iterator has been created. To permit sharing between a tree and an iterator, Malecha and Morrisett use fractional permissions, where we use the H predicate. They work in an axiomatic extension of Coq, whereas our proofs are done in a shallowly embedded program logic, since our programs are written in an existing programming language (Java).

Dinsdale-Young et al. [5] present another approach to reasoning about shared data structures, which gives the client a fiction of disjointness. Roughly speaking, they define a new abstract program logic for each module (they can be combined) for abstract client reasoning. Their approach allows one to give a client specification similar to ours, but without using the H and with the abstract structures (Tree / Snap / Iter) being predicates in the (abstract) program logic. This has the advantage that one can use ordinary framing for local reasoning.

Dinsdale-Young et al. [4] also presented an approach to reasoning about sharing. Sharing can happen in certain regions, and the module implementor has to define a protocol that describes how data in the shared region can evolve. What corresponds to our abstract structures can now be seen as separation logic predicates and thus one can use ordinary framing for local reasoning.

In both approaches [5] and [4] the module implementor has more proof obligations than in our approach: In [5] he must show that the abstract operations satisfy some conditions related to the realization of the abstract structures in the concrete heap. In [4] she must show related properties phrased in terms of certain stability conditions.

Compared to the work of Dinsdale-Young et al., our approach has the advantage that it is arguably simpler, with no need to introduce new separation (or context) algebras for the modules. That is why we could build our formalization on an implementation of standard separation logic in Coq.

Rustan Leino made a solution for a custom implementation of this data structure (A1B1) using Dafny [11]. Dafny verifies that if a snapshot is present, the nodes are shared and not mutated by the tree operations. His solution does not (yet) verify the content of the tree, snapshots or iterators. Our verification specifies the concrete heap layout. Dafny does not support abstract specification due to the lack of inheritance. The trusted code base is different: Dafny relies on Boogie, Z3 and the CLR, whereas our proof trusts Coq.

7 Conclusion and Future Work

We have presented snapshotable trees as a challenge for formalized modular reasoning about mutable data structures that use sharing extensively, and given an abstract specification of the ITree interface. Moreover, we have presented a formalization of the A1B1 implementation of snapshotable trees.

The overall size of the formalization effort is roughly 5000 lines of Coq code and it takes 2 hours to `Qed` the proofs. This is quite big compared to other formalization efforts of imperative programs in Coq, such as Hoare Type Theory / Ynot [14,15]. The main reason is that we are working in a shallowly embedded program logic for a Java-like language, whereas Hoare Type Theory / Ynot is an axiomatic extension of Coq. Thus our formalization includes both the operational semantics of the Java subset and the soundness theorems for the program logic; also, Java program variables cannot simply be represented by Coq variables.

We also plan to verify the even subtler implementations A1B2, A2B1 and A2B2, which are expected to provide further insight into the challenges of dealing with shared mutable data and unobservable state changes. Through those more complex applications of separation logic we hope to learn more about desirable tool support, including how to automate the "obvious" reasoning that currently requires much thought and excessive amounts of proof code. Although we have not formally verified these implementations yet, we are fairly certain they would match the interface specification presented in Section 3. In all four implementations the tree is conceptually separate from its snapshots, which is the property required by the interface, and the invariant H allows us to describe the heap layout very precisely, using techniques shown in Section 4.

Finally, we would like to explore how to combine the advantages of our approach and those of Dinsdale-Young's approach discussed above.

References

1. Andersson, A.: Balanced Search Trees Made Simple. In: Dehne, F., et al. (eds.) WADS 1993. LNCS, vol. 709, pp. 60–71. Springer, Heidelberg (1993)
2. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 22–38. Springer, Heidelberg (2011)
3. Biering, B., Birkedal, L., Torp-Smith, N.: BI-hyperdoctrines, higher-order separation logic, and abstraction. ACM Trans. Program. Lang. Syst. 29(5) (2007)
4. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
5. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and Refinement for Local Reasoning. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 199–215. Springer, Heidelberg (2010)
6. Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R.: Making data structures persistent. Journal of Computer and Systems Sciences 38(1), 86–124 (1989)
7. Guibas, L., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th FCS, pp. 8–21. Ann Arbor, Michigan (1978)
8. Kokholm, N., Sestoft, P.: The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen (January 2006)
9. Krishnaswami, N.: Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic. PhD thesis. Carnegie Mellon University (forthcoming, 2011)
10. Krishnaswami, N.R., Birkedal, L., Aldrich, J.: Verifying event-driven programs using ramified frame properties. In: TLDI, pp. 63–76. ACM (2010)
11. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
12. Malecha, G., Morrisett, G.: Mechanized verification with sharing. In: 7th International Colloquium on Theoretical Aspects of Computing (September 2010)
13. Mehnert, H.: Kopitiam: Modular Incremental Interactive Full Functional Static Verification of Java Code. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 518–524. Springer, Heidelberg (2011)
14. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: Hook, J., Thiemann, P. (eds.) Proc. of 13th ACM ICFP 2008, pp. 229–240. ACM (2008)
15. Nanevski, A., Vafeiadis, V., Berdine, J.: Structuring the verification of heap-manipulating programs. In: Proceedings of POPL (2010)
16. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL, pp. 247–258 (2005)
17. Petersen, R.L., Birkedal, L., Nanevski, A., Morrisett, G.: A Realizability Model for Impredicative Hoare Type Theory. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 337–352. Springer, Heidelberg (2008)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: IEEE Proc. of 17th Symp. on Logic in CS (November 2002)

19. Sedgewick, R.: Left-leaning red-black trees, http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf
20. Svendsen, K., Birkedal, L., Parkinson, M.: Verifying Generics and Delegates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 175–199. Springer, Heidelberg (2010)

A Appendix

We define here the ITree and the Iterator interface specification as Coq definitions, as well as the Stack class. We use the name *SPred* for the finite set of abstract structures containing exactly one Tree structure and any number of Snap and Iter structures.

The notation $\widehat{f}$ lifts the function f such that it operates on expressions rather than values.

A detailed explanation of the notation and of lifting can be found in [2].

$$
\begin{aligned}
&\mathit{ITree} \triangleq\ \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}.\\
&\quad (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{contains}(\texttt{this}, \texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \in \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{contains}(\texttt{this}, \texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this}, \tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this}, \tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \in \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{add}(\texttt{this}, \texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \{\texttt{x}\} \cup \tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \notin \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{snapshot}(\texttt{this}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this}, \tau), \widehat{\mathsf{Snap}}(\texttt{r}, \tau)\} \uplus \phi)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{iterator}(\texttt{this}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this}, \tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \exists IC : \mathit{classname}.\ \mathit{Iterator}\ IC\ H \wedge \texttt{r} : IC \wedge\\
&\qquad\quad \widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this}, \tau), \widehat{\mathsf{Iter}}(\texttt{r}, [\tau])\} \uplus \phi)\})\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \implies v : C) \wedge (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \implies v : C))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \vdash H(\phi)))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Iter}(v, \alpha)\} \uplus \phi) \vdash H(\phi)))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau, \tau' : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad \tau = \tau' \implies (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(v, \tau')\} \uplus \phi)))
\end{aligned}
$$

$$\begin{array}{l}
\mathit{Iterator} \triangleq \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}. \\
\quad (\forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{hasNext}(\mathtt{this}) \mapsto \\
\qquad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \land \mathtt{r} = (\alpha \neq \mathsf{nil})\}) \\
\quad \land (\forall x : \mathbb{Z}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{next}(\mathtt{this}) \mapsto \\
\qquad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, x{::}\alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \land \mathtt{r} = x\})
\end{array}$$

$$\begin{array}{l}
\mathit{Stack\_spec} \triangleq \forall T : \mathit{Type}. \\
\quad \exists SR : \mathit{classname} \to (\mathit{val} \to T \to \mathit{HeapAsn}) \to \mathit{val} \to T^* \to \mathit{HeapAsn}. \\
\quad (\forall C : \mathit{classname}.\ \forall P : \mathit{val} \to T \to \mathit{HeapAsn}. \\
\qquad \mathtt{Stack{::}new}() \mapsto \{\top\}\_\{\mathtt{r}.\ \widehat{SR}\ C\ P\ \mathtt{r}\ \mathsf{nil}\} \\
\quad \land (\forall \alpha : T^*.\ \mathtt{Stack{::}empty}(\mathtt{this}) \mapsto \\
\qquad \{\widehat{SR}\ C\ P\ \mathtt{this}\ \alpha\}\_\{\mathtt{r}.\ \widehat{SR}\ C\ P\ \mathtt{this}\ \alpha \land \mathtt{r} = (\alpha = \mathsf{nil})\}) \\
\quad \land (\forall \alpha : T^*.\ \forall t : T.\ \mathtt{Stack{::}push}(\mathtt{this}, \mathtt{x}) \mapsto \\
\qquad \{\widehat{SR}\ C\ P\ \mathtt{this}\ \alpha * \widehat{P}\ \mathtt{x}\ t \land \mathtt{x} : C\}\_\{\widehat{SR}\ C\ P\ \mathtt{this}\ (t :: \alpha)\}) \\
\quad \land (\forall \alpha : T^*.\ \forall t : T.\ \mathtt{Stack{::}pop}(\mathtt{this}, \mathtt{x}) \mapsto \\
\qquad \{\widehat{SR}\ C\ P\ \mathtt{this}\ (t :: \alpha)\}\_\{\mathtt{r}.\ \widehat{P}\ \mathtt{r}\ t * \widehat{SR}\ C\ P\ \mathtt{this}\ \alpha\}) \\
\quad \land (\forall \alpha : T^*.\ \forall t : T.\ \mathtt{Stack{::}peek}(\mathtt{this}, \mathtt{x}) \mapsto \\
\qquad \{\widehat{SR}\ C\ P\ \mathtt{this}\ (t :: \alpha)\}\_\{\mathtt{r}.\ \widehat{P}\ \mathtt{r}\ t * \\
\qquad\ (\forall u : T.\ \widehat{P}\ \mathtt{r}\ u \mathrel{-\!\!*} \widehat{SR}\ C\ P\ \mathtt{this}\ (u :: \alpha))\})) \\
\quad \land (\forall C : \mathit{classname}.\ \forall P, P' : \mathit{val} \to T \to \mathit{HeapAsn}. \\
\qquad (\forall v : \mathit{val}.\ \forall t : T.\ (P\ v\ t \vdash P'\ v\ t)) \implies \\
\qquad\quad \forall v : \mathit{val}.\ \forall \alpha : T^*.\ (SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha))
\end{array}$$

Comparing Verification Condition Generation with Symbolic Execution: An Experience Report

Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff

ETH Zurich, Switzerland
{ioannis.kassios,peter.mueller,malte.schwerhoff}@inf.ethz.ch

Abstract. There are two dominant approaches for the construction of automatic program verifiers, *Verification Condition Generation* (VCG) and *Symbolic Execution* (SE). Both techniques have been used to develop powerful program verifiers. However, to the best of our knowledge, no systematic experiment has been conducted to compare them.

This paper reports on such an experiment. We have used the specification and programming language Chalice and compared the performance of its standard VCG verifier with a newer SE engine called Syxc, using the Chalice test suite as a benchmark. We have focused on comparing the efficiency of the two approaches, choosing suitable metrics for that purpose. Our metrics also suggest conclusions about the predictability of the performance. Our results show that verification via SE is roughly twice as fast as via VCG. It requires only a small fraction of the quantifier instantiations that are performed in the VCG-based verification.

1 Introduction

During the last years, automated program verification has progressed significantly. This progress is due to advances in each of the three layers that comprise an automatic verification tool: the specification methodology, the program verifier, and the theorem prover. The *program verifier* layer extracts proof obligations from the specified program and passes them to the theorem prover. There are two prevalent approaches to design program verifiers:

- *Verification Condition Generation* (VCG) uses programming calculi such as weakest preconditions [9] to compute a formula whose validity entails the correctness of the program. VCG is used in many state-of-the-art verifiers, in particular, those built on top of the intermediate languages Boogie [18] and Why [10]. Examples of VCG-based verifiers are Chalice [21], Dafny [17], ESC/Java [7], Frama-C [1], Spec# [3], and VCC [8].
- *Symbolic Execution* (SE) [14] executes a program using symbolic instead of concrete values and accumulates constraints on those values, which are used to generate proof obligations. SE has gained momentum especially in the separation logic world (for instance, jStar [24], Smallfoot [5], and VeriFast [12]), but has also been used in combination with other specification methodologies, for instance in KeY [4] and Syxc [28].

To the best of our knowledge, there has been no systematic comparison of the two approaches. Such a comparison would provide insights into the workings of current verification tools and useful guidance for the design of future ones. This guidance is especially important since the choice of specification methodology no longer dictates a verification approach. For instance, dynamic frames are supported in the VCG-based Dafny and the SE-based KeY [27]. Permission-based methodologies such as separation logic are supported in the VCG-based Chalice and the SE-based Syxc; VeriCool [29] supports both VCG and SE in a single tool.

This paper reports on an experiment we conducted to compare verification condition generation and symbolic execution. We used the Chalice language [21] and compared its standard VCG-based verifier [20] to a new SE engine [28]. Both verifiers use Z3 [22] as theorem prover. Chalice is an interesting target for such a comparison for several reasons: (1) It is small enough to make it feasible to develop two verifiers for the same language. (2) It provides a variety of features that make verification challenging, in particular, dynamic memory allocation and multi-threading. (3) Chalice's specification methodology is closely related to separation logic [25] such that our observations can be expected to apply also to verifiers for that methodology.

We ran both verifiers on the Chalice test suite. Our experiment focuses mainly on the efficiency of the verification by measuring run times. However, we also measured the number of quantifier instantiations performed by Z3, which gives an impression of how predictable the performance is, as well as the number of conflicts encountered during the verification, which characterizes the size of the search space. The results of these measurements are reported in Sec. 3. They show that the SE-engine Syxc is about twice as fast as the VCG-based Chalice verifier on most benchmarks. It requires much less quantifier instantiations and generally leads to fewer conflicts. Further comparisons, for instance, about the ease of understanding verification failures, are left as future work.

The initial comparisons of the performance of the two verifiers identified several outliers. Their investigation lead to interesting observations about the design of Syxc, which we discuss in Sec. 4.

2 Background

In this section, we provide the background on verification condition generation and symbolic execution that is used in the rest of the paper. We assume for both approaches that loops are annotated with loop invariants.

2.1 Verification Condition Generation

Verification condition generators use a programming calculus such as a weakest precondition calculus to compute a *verification condition* (VC), a pure logical formula whose validity entails the correctness of the program w.r.t. its specification. The VC is then fed to the theorem prover. Many modern program verifiers generate verification conditions by first translating the program and its specification into an intermediate language such as Boogie [18] or Why [10] and then computing the VC on the intermediate representation.

Since a verification condition is a pure logic formula, it must include all knowledge that is needed to prove the correctness of the program. This knowledge includes many properties of the programming language semantics, for instance, about values and types. For imperative languages, it also contains a heap model, typically expressed as a global map from locations to values. All aspects of the verification, including reasoning about heap properties such as aliasing, are then left to the theorem prover.

A characteristic of the VCG approach is that it computes only one VC per module and, thus, invokes the theorem prover only once per module. On the upside, dealing with one large VC allows the theorem prover to apply optimizations. On the downside, the VC tends to be large, even for small programs, which increases the complexity for the theorem prover and complicates quantifier instantiation because the prover will in general find more matching patterns. Another drawback of large VCs is that they are undecipherable to the human, so VCG-based debugging needs extra tool assistance [23].

2.2 Symbolic Execution

Symbolic execution [14] verifies a program by simulating an execution with variables that do not take concrete values, but whole expressions (known as *symbolic values*). Knowledge about the symbolic values is accumulated in a logical formula called the *path condition*. When a branch in the control flow is encountered, the execution takes both branches and conjoins the appropriate formula to the path condition of each branch. The prover is called each time an assertion needs to be proved. The prover is then given the path condition as an assumption and a relatively simple proof obligation.

A known limitation of this approach is its exponential execution time in the number of branches. Since SE verifies each path through a program independently, this may cause inefficiencies when there are redundancies between the proof obligations for different paths. With VCG, one would hope that theorem provers detect and exploit such redundancies in the single large verification condition, avoiding or moderating the exponential blowup of SE.

Berdine et al. [5] have noticed that the heap topology described by a separation logic specification can be treated independently of the rest of the information that the formula contains. This observation makes it possible for an SE-verifier to deal with heap properties inside the verifier, *without sending them to the prover*. This approach is implemented in Smallfoot and has been adopted by VeriCool, VeriFast, as well as the verifier Syxc that we used for our experiment. Smallfoot-style symbolic execution produces proof obligations that are simpler than VCs, because more reasoning is done within the program verifier and less in the theorem prover.

A potential drawback of this division of labor is that the prover has only partial access to the information that is available to the verifier, for instance, because heap properties are not encoded in the proof obligation sent to the prover. This may mean that the prover can prove less than in the VCG approach.

Understanding the implications of Smallfoot-style SE in terms of efficiency, predictability, and completeness was one of the motivations for our experiment. While building Syxc, we encountered several instances of the incompleteness problem, and a striking instance of the exponential branching problem, which we report on in Sec. 4.

3 Experiment and Results

In this section, we describe the set-up of our experiment and discuss our measurements. The tools and the benchmarks that were used in the experiments can be found on-line under `http://www.pm.inf.ethz.ch/people/schwerhoffm/vstte_sevcg_verifiers.zip`

Benchmarks. For the experiment we have utilized 29 test cases from the current Chalice test suite. These test cases exercise the main Chalice features such as fractional permissions, objects, threads, locks, and message passing. The development of the Chalice test suite was unrelated to the present experiment. We separated the test cases into correct and incorrect programs. Both tools verify all the correct programs and reject all incorrect programs.

The Chalice test suite includes additional examples that we could not benchmark, because Syxc does not yet support all features the Chalice language offers. For five examples we also had to (1) manually desugar certain expressions because they are not supported by Syxc, and to (2) remove a few methods that used unsupported features.

Verification in Chalice is modular, that is, each method is verified without considering its callers or the implementations of methods it calls. Therefore, the total size of the benchmark programs is less important than the size of individual methods. The largest two programs are two implementations of AVL trees, one iterative and one recursive. The recursive version is by far the largest benchmark in terms of lines of code and number of methods, whereas the iterative version includes the longest method.

Verification Tools. The VCG-based Chalice verifier used in our experiments is built from revision 08870c66a385. This is a slightly outdated version that does not use the new Chalice permission model [11], which is not yet implemented in Syxc. It uses the Boogie build from revision ba07abf9500e and Z3 3.1 x64. Boogie has been limited to a single error per Boogie procedure, since this comes closest to the behavior of Syxc, which is limited to one error per Chalice method. Both tool chains use the SMTLib2 front-end of Z3 and log their interactions with Z3. In this configuration Z3 is used via std-io, not via an API.

The Syxc tool chain comprised the same Chalice build in order to parse the input, and the same Z3 installation. It uses Z3 in a configuration that is nearly identical to that of Boogie, with two minor differences: (1) Z3 responds to every command it receives, and not only to satisfiability checks, and (2) declarations are global instead of scope-local to optimize the verification of multiple paths.

Both the standard VCG-based Chalice verifier and the SE engine Syxc are written in Scala 2.8.

Two possibly relevant differences between the Z3 encoding generated by Boogie and by Syxc are (1) Syxc currently uses a "weak" Z3 type system, in the sense that there are not types (sorts) for snapshots, references and lists, all of which are encoded as integer-typed symbols. (2) Syxc uses the same sequence axioms as Boogie, except that they only range over integers, whereas Boogie's sequence axioms are polymorphic. Sequences are used by six out of the 22 examples in our test suite.

The Metrics. In the experiment, we measured for each benchmark program:

– Verification time (wall time) in seconds
– Number of quantifier instantiations that Z3 performed during the verification
– Number of conflicts encountered by Z3 during the verification time

Verification time is the total run time, including parsing and type checking, symbolic execution or verification condition generation, and time spent in the prover. Since Chalice and Syxc use the same parser and type checker, differences in the measurements can be attributed to the actual verification.

The number of quantifier instantiations is interesting, because it serves as an indication of the predictability of the verifier. Quantifier instantiation is guided by heuristics, which are often overly sensitive to small changes in the program or specification. The fewer quantifier instantiations a technique needs, the less it depends on heuristics.

A conflict is a failed attempt by the prover to assign a value to a variable. More instantiation attempts indicate that the prover had to explore a larger search space before finding a satisfying assignment.

Experiment. The experiments have been run on an Intel Core2 Quad CPU Q9550 2.83GHz, 4GB RAM, with Windows 7 Enterprise x64. For each benchmark program, the statistics have been collected by verifying the program with the VCG-based Chalice verifier and with Syxc, and then running Z3 on the interaction log file in order to get statistics from Z3. This has been done ten times for each file and each verifier. The numbers of these ten runs have been averaged. The run times of both verifiers have been measured with the same tool.

Results. The results of the experiment are summarized in Fig. 1. For each program, we show: lines of code, number of methods, the results of SE, the results of VCG, and finally, for each metric, the percentage of the result of SE over the result of VCG. Times are measured in seconds. Incorrect programs are in folder "fail". Correct programs are in folder "hold".

The standard deviation of the averages (not shown in the table) is negligible in all measurements. For run time, the worst deviation is 3% of the mean time, observed for the fail\cell example in the VCG tool. For the other metrics, VCG

File			Syxc			Chalice			%		
Name	LOC	Methods	Time	QI	Conflicts	Time	QI	Conflicts	Time	QI	Conflicts
fail\cell	30	3	1.12	13	1	2.39	410	21	46.86	3.17	4.76
fail\LoopLockChange	44	3	1.12	134	9	2.47	1519	90	45.34	8.82	10.00
fail\ProdConsChannel	60	6	1.24	47	10	2.51	1009	72	49.40	4.66	13.89
fail\prog1	53	7	1.12	28	6	2.41	1045	97	46.47	2.68	6.19
fail\prog2	26	4	1.04	7	3	2.25	124	12	46.22	5.65	25.00
fail\prog3	20	3	0.98	8	1	2.25	128	7	43.56	6.25	14.29
fail\prog4	36	3	1.09	25	4	2.35	375	47	46.38	6.67	8.51
hold\AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
hold\AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
hold\cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
hold\CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
hold\CopyLessMessagePassing-with-ack	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
hold\CopyLessMessagePassing-with-ack2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
hold\dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
hold\iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
hold\iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
hold\LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
hold\OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
hold\PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
hold\ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
hold\producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
hold\quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
hold\RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
hold\Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
hold\swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
hold\prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
hold\prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
hold\prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
hold\prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07

Fig. 1. Results of the Experiment

has no deviation (deterministic behavior), while SE has very minor deviations only in the large examples (these are attributed to the non-determinism of some standard Scala collection libraries). The insignificant deviation indicates that the results are stable and repeatable.

A consistent observation is that the execution time of SE is in the range of 40% to 50% of that of VCG. Thus, SE outperforms VCG, by what seems to be a constant factor.

An interesting observation is that the performance benefit of SE in the two largest programs (the AVL tree implementations) is bigger: there, the execution time of SE drops to around 20% of the time of VCG. This is promising, but far from conclusive for the scalability of SE. To draw reliable conclusions about scalability, we would need more and larger benchmarks. However, we chose not to develop extra benchmarks just for this experiment.

The number of quantifier instantiations in SE is consistently under 10% of that in VCG. Although it was expected that less quantifier instantiations would occur in SE because heap properties are handled outside the prover, such a difference was beyond our expectations. This is an indication that the performance of SE might be more predictable than the performance of VCG.

The percentage of conflicts had a greater variation than the percentage in the other metrics. Still conflicts in SE are consistently much fewer than in VCG, indicating that SE verification is "more focused" (explores a smaller proof space) than VCG.

We did not find significant differences in the results between correct and incorrect benchmark programs, although we intuitively believed that SE would deliver verification errors earlier. That said, given the small number of failing test cases that we have, this observation should be taken with a grain of salt.

Among the test cases, there is one outlier, the "iterator" program, in which VCG marginally outperforms SE. We are currently looking into this outstanding behavior, with the hope of discovering interesting design issues, as happened with outliers in previous experiments (see for example Sec. 4.2).

4 Additional Observations

In Sec. 2.2, we mentioned two challenges for symbolic execution: (1) the potential incompleteness caused by separating heap properties from path conditions and reasoning about the former in the verifier and about the latter in the prover (in Smallfoot-style SE), and (2) the exponential branching problem. The comparison to the VCG-based Chalice verifier exposed problems related to both challenges in an earlier version of Syxc: Syxc was not able to prove some examples that Chalice proved soundly, and for one example, the relative performance of Syxc was significantly worse than for the other examples. In this section, we discuss the problems and describe the solution that is implemented in the current version of Syxc.

4.1 Heap Compression

Smallfoot-style SE is more susceptible to incompleteness than VCG, since the SE verifier does not give all the available heap-related information to the theorem prover. Dealing with such incompletenesses is not straightforward and involves several design trade-offs. In this subsection, we show one such incompleteness and its solution.

In Smallfoot-style SE, information about a heap location is expressed by a *heap chunk* $t.f \mapsto t'$, where t, t' are symbolic values and f is a field name. The heap chunk $t.f \mapsto t'$ means that access to the field f of the object t is granted and that t' is the value of that field. Syxc adds fractional permissions, so a Syxc heap chunk has the following form: $t.f \mapsto t' \# p$. The extra information p is the percentage of permission granted. Heap chunks are stored in the *symbolic heap*, which is used by the verifier to decide access permissions, ideally without consulting the prover. The relationship between symbolic values is encoded in the path condition, which is available to the prover when a proof obligation is checked.

Suppose that a program acquires the heap chunks $t_1.f \mapsto t_2\#30\%$ and $t_3.f \mapsto t_4\#30\%$. Suppose also that the path condition implies $t_1 = t_3$. This situation illustrates two sources of incompleteness caused by the division of labor between the verifier and the prover. First, the verifier fails to show an assertion that we have 60% permission to $t_1.f$, because without consulting the prover, it cannot derive the needed information $t_1 = t_3$ from the path condition. Second, the prover fails to show an assertion that $t_2 = t_4$, because this equality is a consequence of the contents of the symbolic heap, which is not available to the prover.

We fix this problem by using the path condition and the theorem prover to *compress* the symbolic heap, that is, to reflect aliasing information as follows: For each pair of heap chunks $t_1.f \mapsto t_2\#p_1$ and $t_3.f \mapsto t_4\#p_2$:

- Ask the prover if $t_1 = t_3$ follows from the path condition.
- If yes, then:
  - Replace the two heap chunks by one: $t_1.f \mapsto t_2\#p_3$
  - Add to the path condition the following conjuncts: $t_1 = t_3$, $t_2 = t_4$, $p_3 = p_1 + p_2$

Notice that the compression process introduces more equalities into the path condition. The stronger path condition may justify more compression. Therefore, we compress the symbolic heap iteratively, an operation that costs $O(n^3)$ queries to the prover (where n is the number of heap chunks).

A related problem appears when we have the following two heap chunks: $t_1.f \mapsto t_2\#60\%$ and $t_3.f \mapsto t_4\#60\%$. Since the accumulative permission to a field cannot exceed 100%, we can conclude from this symbolic heap that $t_1 \neq t_3$. To allow the prover to exploit this knowledge, we add it to the path condition during the heap compression.

The need for heap compression is due to the fact that Smallfoot-style SE does not include heap (and permission) properties in the proof obligations sent to the prover, while they are included in the verification condition produced by a VCG-based verifier. Our experiments show that SE outperforms VCG even though we have to perform the explicit heap compression. Nevertheless, we believe that it has a lot of potential for optimization, for instance, it could be performed on demand only when a proof fails.

Our experience with VCG and SE suggests that the higher efficiency of SE does not necessarily come at the price of less precision. However, tool developers have to be careful to enable the necessary information flow between the symbolic heap and the path condition.

4.2 Branching

Symbolic execution verifies each path through a module separately. In addition to the branches introduced by control flow, Syxc also introduces branches when it needs to represent properties expressed as implications. For example, the Chalice specification $b \Rightarrow acc(e.f)$ means that if b is true then the current thread has write access to field $e.f$, and no access permission otherwise. Since such a conditional permission cannot be represented with the heap chunks described above, Syxc branches. One branch adds b to the path condition and an appropriate heap chunk for $e.f$ to the symbolic heap, whereas the other branch adds $\neg p$ to the path condition and leaves the symbolic heap unchanged.

Experiments with an earlier version of Syxc identified "Peterson's Algorithm" from the Chalice test suite as a significant outlier: while SE performed better than VCG in the other examples, it performed significantly worse here. Our analysis revealed that the bad performance was caused by excessive branching on implications. However, all the implications in this example were *pure*, that is, did not contain any access permission predicates. Therefore, the whole implication could be added to the path condition, and no branching is necessary. Implementing this change made SE outperform VCG again.

Our experience suggests that the theorem prover has better ways to handle implications than just case splits; the larger proof obligation with the implications allows the prover to avoid redundant proofs. This seems to confirm the intuition that VCG outperforms SE in case of heavy branching.

Note that our solution is possible only for pure implications, because a pure implication does not influence the symbolic heap. To alleviate the problem of branching on impure implications, we consider introducing "conditional" heap chunks. Initial experiments in this direction show some promise.

5 Related Work

As mentioned earlier, there is a multitude of tools that support SE in the style of Smallfoot [5]. These include jStar [24], VeriCool [29], VeriFast [12], and Syxc [28]. Of those, Smallfoot, jStar, and VeriFast use a fragment of separation logic [26] as their specification language, while the target programming language differs. VeriCool and Syxc use implicit dynamic frames [29]. VeriFast and Syxc also support fractional permissions [6]. Implicit dynamic frames have similar expressive power to the fragment of separation logic used by SE tools [25].

The KeY system [4] uses a different form of symbolic execution that does not separate heap properties from "pure" properties, in the style of Smallfoot. The specification language of KeY is JML [16] with dynamic frames [13]. The KeY approach admits some degree of interaction with the prover. Our comparison focuses on Smallfoot-style SE. It is not clear to what degree our observations apply to KeY-style SE.

Tools that are based on VCG include Chalice [20], Dafny [17], ESC/Java [7], Frama-C [1], Regional Logic [2], Spec# [3], VCC [8], and VeriCool [29]. Most of them employ the intermediate languages Boogie [18] or Why [10]. Chalice and VeriCool support implicit dynamic frames (in Chalice with fractional permissions), Dafny supports dynamic frames, and Spec# and VCC support ownership [19].

VeriCool [29] supports both SE and VCG. The authors report on experiments using the visitor pattern and an artificial example that show an overwhelming advantage for SE [30]. Our experiments do not confirm this result, even though our tools are similar to VeriCool (based on implicit dynamic frames, Smallfoot-style SE, and Z3). We decided to perform our comparison using Chalice and Syxc rather than VeriCool because the specification language supported by VeriCool's VCG engine is significantly different from the one supported by its SE engine, which means that we could not have tried the exact same code in both engines. For example, the VeriCool specification language for SE does not support quantification.

VSTTE 2010 hosted a software verification competition [15]. The evaluation focused on the quality of specifications and the strength of the verification, but does not permit a comparison of the efficiency of verification approaches like the one we present here.

6 Conclusion

In this paper, we reported on an experiment that compared two dominant approaches for the construction of automated program verifiers, Verification Condition Generation and Symbolic Execution. Our experiment shows that SE is generally more efficient, leads to a smaller search space for the prover, and requires fewer quantifier instantiations. Even though the differences in the run times are noticeable, they are just a constant factor. Future experiments will have to show whether more substantial differences exist for larger modules. Another topic for future work is to compare other important criteria, such as completeness, the ease of understanding and fixing verification errors, as well as the performance on more substantial failing cases. The latter is also a strong indicator of the responsiveness of the tools in an interactive setting, in which the program and its specifications are developed over several iterations of running the tool.

We believe that experimental evaluation is an important aspect of advancing the field of software verification. The VSTTE verification competition provides interesting insights in to the strength of various verifiers (tools and their users). Our experiment complements the competition by providing a comparison of two verification approaches in terms of their efficiency. We hope that it will encourage others to perform additional studies that will help the community to better understand the impact of design decisions on the performance of verification tools.

Acknowledgments. We are grateful to Jan Smans for his patient explanations of VeriCool and to Leonardo de Moura and Christoph Wintersteiger for helpful explanations of Z3's statistics and heuristics. We also like to thank Michał Moskal for help on Boogie and Z3, Uri Juhasz for providing us with the AVL-tree example, and Alex Summers for many fruitful discussions. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Almeida, J.B., Frade, M.J., Pinto, J.S., Melo de Sousa, S.: Verifying C programs. In: Rigorous Software Development. Undergraduate Topics in Computer Science, pp. 241–256. Springer, Heidelberg (2011)
2. Banerjee, A., Naumann, D., Rosenberg, S.: Regional Logic for Local Reasoning about Global Invariants. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
3. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Communications of the ACM 54(6), 81–91 (2011)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334, pp. 69–177. Springer, Heidelberg (2007)
5. Berdine, J., Calcagno, C., O'Hearn, P.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
6. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
7. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskał, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
9. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall Series in Automatic Computation. Prentice Hall (1976)

10. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud (2003)
11. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Fractional permissions without the fractions. In: Formal Techniques for Java-like Programs, FTfJP (2011)
12. Jacobs, B., Smans, J., Piessens, F.: VeriFast: Imperative programs as proofs. In: VS-Tools Workshop at VSTTE 2010 (2010)
13. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
15. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)
16. Leavens, G., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, I., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer (1999)
17. Leino, K.R.M.: Specification and verification of object-oriented software. In: Marktoberdorf International Summer School 2008. Lecture Notes (2008)
18. Leino, K.R.M.: This is Boogie 2. Working Draft (2008), `http://-research.microsoft.com/en-us/um/people/leino/papers.html`
19. Leino, K.R.M., Müller, P.: Object Invariants in Dynamic Contexts. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
20. Leino, K.R.M., Müller, P.: A Basis for Verifying Multi-Threaded Programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
21. Leino, K.R.M., Müller, P., Smans, J.: Verification of Concurrent Programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design V. Lecture Notes In Computer Science, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
22. Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
23. Müller, P., Ruskiewicz, J.N.: Using Debuggers to Understand Failed Verification Attempts. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)
24. Parkinson, M., Distefano, D.: jStar: Towards practical verification for Java. In: Harris, G.E. (ed.) OOPSLA 2008, pp. 213–226. ACM (2008)
25. Parkinson, M., Summers, A.: The Relationship Between Separation Logic and Implicit Dynamic Frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011)
26. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society (2002)

27. Schmitt, P., Ulbrich, M., Weiß, B.: Dynamic Frames in Java Dynamic Logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011)
28. Schwerhoff, M.: Symbolic execution for Chalice. Master's thesis, ETH Zurich (2011)
29. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
30. Smans, J., Jacobs, B., Piessens, F.: Symbolic execution for implicit dynamic frames (2009), `http://people.cs.kuleuven.be/~jan.smans/vericool3/`

Verification of TLB Virtualization Implemented in C[⋆]

Eyad Alkassar[1], Ernie Cohen[2], Mikhail Kovalev[1], and Wolfgang J. Paul[1]

[1] Saarland University, Saarbrücken, Germany
{eyad,kovalev,wjp}@wjpserver.cs.uni-saarland.de
[2] European Microsoft Innovation Center (EMIC), Aachen, Germany
ecohen@microsoft.com

Abstract. Efficient TLB virtualization is a core component of modern hypervisors. Verifying such code is challenging; the code races with TLB virtualization code in other processors, with other guest threads, and with the hardware TLBs, and implements an abstract TLB that races with other abstract TLBs and guest threads. We give a general methodology for verifying virtual device implementations, and demonstrate the verification of TLB virtualization code (using shadow page tables) in the concurrent C verifier VCC. To our knowledge, this is the first verification of any kind against a realistic model of a modern hardware MMU.

Keywords: Shadow Page Tables, TLB, Hardware/Software Interaction, Automatic Verification, Virtualization.

1 Introduction

A major challenge in the formal verification of low-level system software is dealing with devices. Because devices run in parallel with software, they typically necessitate concurrent program reasoning even for single-threaded software. Existing verifications of software that interacts with devices (e.g., [3]) rely on transforming the software and hardware into a single transition system. Similarly, verifying correct device virtualization requires reasoning about simulation, which is usually viewed as a relation between transition systems (rather than a property of a single program), and so likewise depends on treating the entire program as a big transition system, an approach sufficient for toy programs but hardly appetizing for industrial-scale software verification.

In this paper, we show how software that interacts with and simulates devices can be verified directly in VCC [4], a verifier for concurrent C. VCC provides several features that are central to our methodology:

– it supports verification of programs with fine-grained concurrency, which is generally needed for programs that deal with devices,

[⋆] Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008.

- it provides ghost state, which we use to represent the abstract state of a virtual device,
- it provides two-state object invariants (i.e., invariants that constrain not only the states of an object but its atomic state transitions also), which we use to express the desired behavior of the virtual device,
- it provides ghost code, which we use to witness the existence of simulation-preserving updates to the abstract state, allowing forward simulation to be expressed using ordinary program reasoning.

We apply our methodology to verify the (partial) correctness of C code (the hypervisor) that virtualizes the MMU of an x64-like processor[1]. The purpose of such virtualization is to provide, to several virtual machines (VMs), each running its own operating system, the illusion that every VM is running alone on a physical machine, even though different machines might try to configure their page tables to use the same physical addresses. To provide this illusion, the algorithm provides an additional level of address translation beyond what is provided by the hardware MMU. It does this by maintaining a separate set of *shadow page tables* (SPTs) for each VM, each SPT shadowing a page table of the guest (GPT). These SPTs are the tables actually used by the hardware for address translation, but are kept invisible to the VMs.

The guest TLB-controlling instructions, such as TLB invalidation (`INVLPG`) or modification of control registers (e.g., *CR3* register), are intercepted by hardware and virtualized by the hypervisor. When a memory access by the guest results in a page fault ($\#PF$) the hypervisor emulates the steps of the virtual MMU by walking the GPTs, setting access (A) and dirty (D) bits in the GPTs, and caching the translations in the SPTs. A correct SPT algorithm guarantees that the *virtual TLB* (VTLB), provided to the VM by the hardware TLB (HTLB) together with the intercept handlers, behaves accordingly to the hardware specification and provides appropriate translations to the guest running in the VM.

The verification target is interesting for several reasons. First, efficient MMU virtualization is perhaps the trickiest part of building correct hypervisors for modern processor architectures (particularly for machines without hardware SLATs). Second, precise reasoning about MMU behavior is central to the correctness of efficient memory management software; because flushing part or all of a TLB is expensive, memory managers often use clever tricks to avoid flushes whenever possible by allowing the TLBs to drift out of synchronization with the page tables (PTs), and using obscure properties of TLB behavior [5] (e.g., that the MMU will set an accessed bit in a nonterminal page table entry (PTE) atomically with its read of that entry, rather than in a separate memory transaction) to deduce that certain translations are not in the TLB. Third, in spite of the critical importance of MMU behavior, it has never been seriously treated in OS kernel verification. For example, the Verisoft project [3] used a synthetic hardware model without TLBs, while the L4.verified project [6] explicitly

[1] While the proof involves specification of many additional functions and data structures, the top-level result depends only on the definitions of the hardware behavior and the coupling relation between the virtual and concrete states (Section 4).

assumed that the TLBs were kept in sync with the page tables, essentially making the TLBs transparent to software. The similar approach was chosen in the Nova micro-hypervisor verification project [7], which used an abstract model of IA-32 hardware with MMU, but without the TLB.

In the rest of the paper we present an overview of the techniques we used for modelling the concurrent hardware and for verifying the code in VCC, formulate the VTLB correctness, and provide the annotated portions of the code crucial for our verification methodology.

2 Verification Methodology

2.1 VCC Background

VCC is a (hopefully) sound, deductive verifier for concurrent C code. As in many other modern, function-modular software verifiers, each function is specified using preconditions, postconditions, and framing information (writes clauses); when verifying the body of a function, the specifications of functions are used to provide the meanings of function calls. VCC allows *ghost data* of various kinds (locals, fields of data structures, and function parameters); ghost data is typically used to represent the abstract value of a data structure. Ghost data can use mathematical types not provided by C (integers and maps of arbitrary size).

Where VCC most differs from other deductive verifiers is in its treatment of data structures and concurrency. In each state, each object (i.e., each instance of a C structured type) is classified as *open* or *closed*, and has a unique *owner*. Only closed objects and threads can own other objects, and only threads can own open objects. A thread can sequentially access only data in an open object that the thread owns[2]; this guarantees that sequential operations in different threads cannot interfere.

The *volatile* keyword has a special meaning in VCC, identifying the shared data which may be changed in an atomic step without opening an object. VCC allows each object type to be annotated with 2-state invariants specifying how the system state may change (in a single atomic step). The invariant of an object is only required to hold in transitions that begin or end with the object closed. (Each type implicitly includes invariants stating that non-volatile fields of the object do not change.) Unlike other verifiers, VCC object invariants can mention arbitrary parts of the state, necessitating the following check: a transition is said to be *legal* if it satisfies the invariants of all objects that it modifies; an object invariant is said to be *admissible* iff it is guaranteed to be preserved by all legal transitions that do not update the object. VCC requires all object invariants to be admissible; this check is made on the basis of type definitions (without looking at code).

A typical idiom is for an invariant of an observer object to assert a property of a subject object, this invariant holding because it was true at the time the

[2] It can also sequentially read data that it knows (by virtue of object invariants) not to be changing.

observer was formed and it is maintained by the subject. Because the subject cannot be relied on to maintain any invariant when it is open, the observer invariant must also guarantee that the subject stays closed. For this invariant to be admissible, the subject must include an invariant that once closed, it remains closed until there are no closed observers observing it. This idiom is very common and useful (particularly for ghost objects), so VCC provides syntactic support for it: subjects supporting such observers are said to be *claimable*, and the supported observers are called *claims*. Each claimable object maintains a count of the number of outstanding claims on it; the object cannot be opened when the claim count is nonzero. When forming a claim that claims a set of subjects and a particular property of the state, VCC does the corresponding admissibility check “inline” to avoid having to introduce a separate object type.

One way to make an observer invariant admissible is for the subject invariant to explicitly require a check of the observer invariant whenever the subject changes state in a way that might violate this invariant. We describe this invariant by saying that the observer *approves* such state changes. Approval provides a general technique for semantic subclassing of concurrency in VCC. A type with an invariant that does not use approval corresponds to a “closed” class (w.r.t concurrency), in the sense that clients cannot strengthen the invariant of an object of the type. A type in which certain updates are approved by a client allow the client to effectively strengthen the invariant of the type as needed (to the extent allowed by his approval). This is the norm when implementing concurrent abstract data types. Approval of a volatile field of an object by a thread that owns the object has the effect of making the field sequential from the standpoint of the owning thread (except that it must still update the field with atomic actions).

2.2 Modelling Hardware

When verifying a multi-threaded program that operates on shared data, the normal procedure is to choose invariants on the shared data that are suitably strong to verify the particular program. To specify a multi-threaded system, such as a multiprocessor, we do not know what particular program will be run, and so have to choose a “generic” invariant for the shared data i.e., the strongest invariant maintained by the system. To construct this invariant, we introduce a ghost variable indicating which component of the system is executing at each step, and require that changes of the shared state (in the current verification, the shared memory) are approved by the currently executing component.

There are two natural ways to model hardware devices in VCC. One is to model a device with a program that exhibits all possible behaviors of the device. This is convenient for verifying a program designed to run in parallel with the device; the verification of the device code guarantees that the invariants required by the program are not so strong as to be potentially broken by actions of the hardware.

The other way to model a device is by modelling the device as an object, where the 2-state invariant of the object gives the allowed transitions of the hardware. This model is convenient for showing correct simulation of the device: we model the virtual device state with ghost data, and show that each update to this data (along with updates to concrete data, like shared memory) satisfies the 2-state invariant of the hardware. (In addition, we normally maintain a 1-state coupling invariant between the concrete state and the virtual state, which has to be maintained by updates to either concrete or virtual state.) In principle, an object model could be used in place of the hardware thread, but there are technical difficulties in doing this that go beyond the scope of this paper, so the current verification uses both models.

3 Specification

The type of n-bit strings $\{0,1\}^n$ is denoted by $\mathbb{B}_n$. We interpret a string $a \in \mathbb{B}_{64}$ either as a 64-bit string, a natural number, or a PTE. We consider a quad-word (64 bits) addressable memory, 45-bit long (quad-word) virtual addresses (VAs), and (quad-word) physical addresses (PAs) 49 bits long (which correspond to architecturally defined byte addresses of 48 bits for virtual and 52 bits for physical addresses). We call the top-most 36 bits (for the VAs) or 40 bits (for the PAs) the page frame number (PFN). We use the operator $=_l :: \mathbb{B}_{36} \mapsto \mathbb{B}_{36} \mapsto \mathbb{B}$ to compare the bits $[35 : 9 \cdot l]$ of two virtual PFNs.

3.1 Host Hardware Model

We model an x64 multi-core/multi-processor machine in the AMD64 style [1] with the record $h :: \mathit{x64conf}$. Shared memory of the system is denoted by $h.mm :: \mathbb{B}_{49} \mapsto \mathbb{B}_{64}$. The hardware configuration of a processor $h.p[i]$ consists of a register *CR3* giving the address of the root PT, a (processor local) TLB *tlb* tagged with *address space identifiers* (ASIDs), a register *asid* providing the tag of the current address space running on the processor, and an uninterpreted variable *state* encapsulating the rest of the processor state. In order to implement and specify the TLB lazy flushing mechanism (which exploits TLB tags), we introduce the function $\mathit{asid}_{gen}(p)$, returning the *generation* of the tags which are still valid in the TLB.

A single PT occupies one memory page (4KB long) and consists of 512 PTEs, each being a 64-bit long union containing the page frame number *pfn* at which the entry is pointing (40-bits long), accessed (A) and dirty (D) bits a and d, a present bit p, and the bits denoting access rights. To simplify reasoning, we introduce the set of access rights r where e.g., $pte.r[rw]$ denotes the write access bit $pte.rw$, with operations of rights comparison and rights restriction (i.e., addition):

$$r_1 \leq r_2 \overset{\text{def}}{=} \forall j : r_1[j] \leq r_2[j] \qquad r_1 + r_2 \overset{\text{def}}{=} \lambda i : r_1[i] \wedge r_2[i] \ .$$

We model the TLB state as a set of page table *walks*, each of which summarizes a partial or complete traversal of the page tables for a given VA. Each walk is given by a virtual PFN *vpfn*, a level l giving the number of page table levels remaining to be walked[3], the page frame number *pfn* of the next page table to be used for translation, the tag *asid* of the address space where the walk was performed and a set r of access rights giving all rights not denied by the walk gathered thus far. A walk is *complete* if its level is 0, and *partial* otherwise.

For the transition system of the host TLB we use the non-deterministic TLB model from [2] extended with ASIDs. We distinguish between autonomous MMU steps, s.t. walk creation, extension, deletion and setting of A and D bits and the TLB-dependent abstracted processor steps, s.t. address invalidation (`INVLPG` and `INVLPGA`), writing of *CR3* register and TLB flush. We use the predicate $\mathit{valid\_step}(h, h')$ to denote that the hardware has performed a valid step of the transition system from configuration h to h'.

We assume that the hypervisor is running untranslated and abstract the TLB to have no walks in the zero ASID (in AMD64 the hypervisor code is always executed with ASID equals zero).

3.2 Virtual Hardware Model

A hypervisor provides to each VM (and to the guest code executed in the VM) the illusion of running on its own private hardware. In the paper we provide this illusion for a single VM (this can be easily generalized to multiple VMs mapped to disjoint host memory portions).

The specification model is an abstract (virtual) machine $g :: \mathit{x64conf}$, which resembles a slightly restricted hardware with the virtual memory $g.mm$ and virtual processors (VPs) $g.p[i]$, $i \in [0 \ldots N_p]$. The hardware virtualization features, such as tagged TLB and virtualization instructions, are not available for the VM.

3.3 Memory Virtualization

The memory of the VM is mapped to some region of the memory of the host machine by means of the injective function $\mathit{gpa2hpa} :: \mathbb{B}_{40} \mapsto \mathbb{B}_{40}$, translating guest physical PFN into the host physical PFN. Translations of guest virtual to guest physical addresses are defined by the GPTs, which are located in the memory of the VM and can be modified by the guest without notifying the hypervisor. When the VM is running runs, the hardware TLB does not have access to the GPTs, but rather operates with SPTs allocated and maintained by the hypervisor.

SPT j of VP i is identified by a pair (i, j) and is obtained from the hypervisor memory by the function[4] $\mathit{spt}(i, j)$. The function $\mathit{i2a}(i, j)$ returns the host PFN of the SPT (i, j). We organize the SPTs for each VP as a tree of SPTs, and assign

[3] To simplify the presentation, we do not consider large pages and legacy addressing modes here, so each complete walk goes through exactly four page tables. Also, we do not consider global page translations.

[4] We assume configuration C to be an implicit parameter in all functions dependent on the memory state.

to each SPT a level ranging from 4 (top-level) to 1 (terminal). The entries of non-terminal SPTs point to other SPTs, while the entries of terminal SPTs point to memory of the VM (under the *gpa2hpa* map). The predicate $walks\_to(i, j, px, j')$ denotes that SPT (i, j) points to SPT (i, j'):

$$walks\_to(i, j, px, j') \overset{\text{def}}{=} (spt(i, j)[px].pfn = i2a(i, j')) \ .$$

The index of the top most SPT of VP i is denoted by $iwo(i)$. The predicate $used(i, j)$ checks whether SPT (i, j) is in use by its VP or is free otherwise, the function $l(i, j)$ returns the level of the SPT (i, j) if it is currently used by the VP. The function $vpfn(i, j)$ returns the prefix of the SPT (VA range for the addresses of the walks that might go through to this SPT) and the function $r(i, j)$ provides the accumulated rights from the top-level SPT to the SPT (i, j). The ASID of the VP i is obtained by the function $asid(i)$ and the ASID generation by the function $asid_{gen}(i)$. The predicate $re(i, j)$ checks whether the SPT is *reachable* from the root $iwo(i)$.

Guest instructions and exceptions that operate on the TLBs are intercepted by the hypervisor so that they can be virtualized in the SPTs.

3.4 Correctness of TLB Virtualization

The abstract VM g is implemented on a host machine h running the hypervisor code. The VM abstraction and the implementation are linked by a coupling invariant. Correctness of the hypervisor is established by proving the simulation between the abstract VM and the VM implementation running atop of the hypervisor. More detailed, we have to show that (i) the coupling invariant is maintained and (ii) the host transitions can be abstracted to valid VM steps (i.e. respecting the hardware transition relation). These properties are encoded by the following invariant:

Invariant 1. *Let h and h' be states of the host hardware machine and the coupling invariant holds between h and g. Then it follows*

$$coupling(h, g) \wedge valid\_step(h, h') \implies \exists g' : valid\_step(g, g') \wedge coupling(h', g') \ .$$

For correct TLB virtualization, we have to consider (i) those parts of the coupling invariant related to the TLB and registers used for address translation and (ii) MMU-related steps of the host.

A VTLB, being part of the virtual hardware g, has to correctly simulate every address translation performed by the HTLB (as well as flushes and autonomous TLB steps). Intuitively, this means that when the guest code is performing a memory access to a guest physical address a (i.e., the VTLB of the VM returns address a for this memory operation), the HTLB should return the translated address $gpa2hpa(a)$. To make this possible, we have to couple every complete walk in the HTLB with the respective ones in the VTLB. We do this by linking the VTLB to the two components of the implementation: the HTLB and the SPTs.

Table 1. Main invariants of the SPT algorithm

Invariant name	**Invariant property**
htlb_walks	$w \in h.p[i].tlb \wedge valid(h, i, w.asid) \implies w \in W[i]$
vtlb_walks	$w \in W[i] \wedge w.l = 0 \wedge w.asid = asid(j) \implies hw2gw(w) \in g.p[j].tlb$
running_asid	$h.p[i].asid \neq 0 \implies valid(h, i, h.p[i].asid)$
distinct_asids	$i \neq j \wedge vp2hp(i) = vp2hp(j) \wedge asid(i) = asid(j)$ $\implies asid_{gen}(i) \neq asid_{gen}(j)$
partial_walks	$w \in W[i] \wedge w.l \neq 0 \wedge asid(j) = w.asid \implies w \in rwalks(j)$ $\wedge w \in rwalks(j) \implies w \in W[vp2hp(j)]$
reachability	$re(i, iwo(i)) \wedge (re(i, j) \wedge walks\_to(i, j, px, j') \implies re(i, j'))$
complete_walks	$w \in cwalks(j) \implies w \in W[vp2hp(j)]$
coupling_gwo	$g.p[i].CR3 = gwo(i)$

Formally, we want the coupling invariant to establish the following property over the HTLB:

$$w \in h.p[i].tlb \wedge w.asid = h.p[i].asid \wedge w.l = 0 \implies hw2gw(w) \in g.p[j].tlb \ , \tag{1}$$

where j is the ID of the currently running VP and the function $hw2gw(w)$ transforms a host walk to the respective walk of the VTLB by applying the inverse of the function *gpa2hpa* to *w.pfn*.

However, to make (1) inductive we have to argue about HTLB walks not only in the currently running ASID, but in all ASIDs which could possibly be scheduled to run without a preceding TLB flush. If an ASID a could be scheduled to run on a host processor (HP) i without a flush, we call it *valid* and we define the set of valid ASIDs in the following way

$$valid(h, i, a) \overset{\text{def}}{=} \exists j : vp2hp(j) = i \wedge a = asid(j) \wedge asid_{gen}(h.p[i]) = asid_{gen}(j) \ ,$$

where the function $vp2hp(i)$ identifies the HP on which VP i is scheduled to run.

For a better partitioning of the invariants in data structures (Sec. 4), we introduce the superset $W[i]$, holding all walks possibly residing in the HTLB of HP i. The desired property (1) is now obtained by the invariants *htlb_walks*, *vtlb_walks*, and *running_asid* (Tab. 1) with the help of *distinct_asids*, which ensures the uniqueness of a VP with a given valid ASID.

To maintain *htlb_walks* when the HTLB is extending a walk, we have to define the content of $W[i]$ and argue about all SPTs, which could be walked by the HTLB in a given configuration.

Our algorithm ensures that the HTLB only accesses *reachable* SPTs i.e., those linked in the SPT tree. The set of all partial walks of VP i sitting on reachable SPTs is defined as

$$\begin{aligned} rwalks(i) \overset{\text{def}}{=} \{w \mid & re(i, j) \wedge w.r \leq r(i, j) \wedge w.pfn = i2a(i, j) \wedge w.l = l(i, j) \\ & \wedge w.vpfn =_{w.l} vpfn(i, j) \wedge w.asid = asid(i)\} \ . \end{aligned}$$

Invariant *partial_walks* (Tab. 1) relates partial walks from $W[i]$ with the walks over the reachable SPTs. Invariant *reachability* helps to maintain *partial_walks* when the HTLB extends a walk (going from one reachable SPT to another).

A straightforward way to identify the complete walks in $W[i]$ is to argue about all terminal shadow PTEs (SPTEs) that could have possibly been walked by the HTLB since the last flush [2]. The task however is cumbersome: a single SPT could be reused for shadowing different GPTs without a complete flush of the HTLB. In this case the HTLB could have walked some SPTE twice - before and after it was reused for a new shadowing. In our approach we only keep track of the terminal SPTEs belonging to reachable SPTs, which is enough to justify the new walks added to the HTLB w.r.t the VTLB. Additionally, we make sure that the VTLB (and the set $W[i]$) drops only the walks which are no longer present in the HTLB.

For a walk through a (terminal) SPT (i, j) let $spte = spt(i, j)[w.vpfn[8 : 0]]$. Then the set of complete reachable walks of VP i is defined as

$$\begin{aligned} cwalks(i) \overset{\text{def}}{=} \{w \mid\, & re(i,j) \wedge l(i,j) = 1 \wedge w.r \leq r(i,j) + spte.r \wedge w.l = 0 \wedge spte.p \\ & \wedge w.vpfn =_1 vpfn(i,j) \wedge w.asid = asid(i) \wedge w.pfn = spte.pfn\} \enspace . \end{aligned}$$

Invariant *complete_walks* (Tab. 1) relates the complete walks over the reachable SPTs with the complete walks in $W[i]$.

Finally, invariant *coupling_gwo* couples the *CR3* register of the VM with the guest walk origin, which is necessary for creating a new walk in the VTLB.

4 Implementation and Verification in VCC

We use ghost data to maintain both the state of the virtual hardware and the state of the host hardware other than the memory (Fig. 1). The hardware transition relation is formulated as a 2-state invariant of the hardware data structure. We use the same data types for modeling the host hardware and for the specification of the abstract VM.

The autonomous part of the host hardware state (e.g., HTLB) is modelled with volatile data and is allowed to change non-deterministically. We locate the host hardware state in the ghost memory, but we do allow limited information flow between some of its fields (e.g., registers and TLB) and the concrete program[5]. We do not restrict the memory updates of the host hardware in the transition relation, since that would require approval of the whole VCC memory by the hardware data structure, making memory changes in the code through regular variable assignments impossible. Instead, we allow VCC software invariants to specify memory transitions on C level. As a result, the autonomous hardware is verified as a C thread with the same annotations on type definitions as the main program.

[5] This is done for lack of a dedicated hybrid type capturing implementation state other then the main memory.

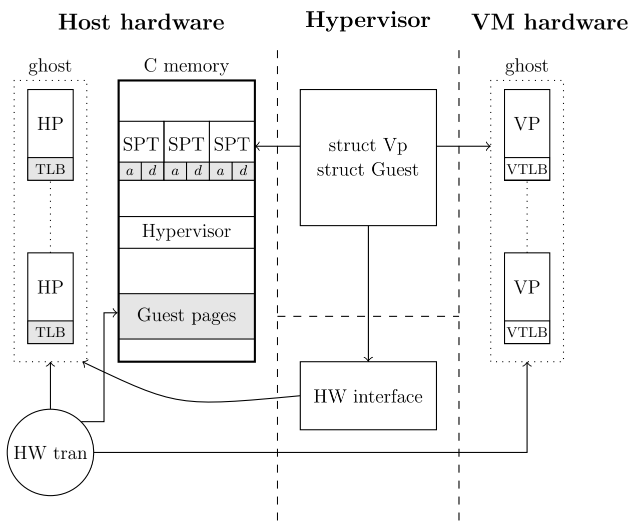


Fig. 1. Approval scenario for the SPT algorithm

The state of the virtual hardware (excluding the memory) is also located in the ghost memory. In contrast to the host hardware, we do specify memory framing for the hardware transitions of the VM. The memory of the VM is abstracted from the portions of the C memory allocated to the machine w.r.t the function *gpa2hpa*. To ensure that every update of the virtual memory is justified by the transition relation of the VM, we model the memory of the VM as volatile data approved by the virtual hardware.

The updates of the virtual hardware, simulating the steps of the VM, are performed by the ghost code in atomic statements, guaranteeing that the transition relation and coupling invariant are maintained by every update. When the step of the virtual hardware involves accessing the implementation memory (e.g., fetching of a GPTE by the $\#PF$ handler), the update to the virtual configuration is done in the same atomic block as the memory access. This allows to simulate a step of the VM on the virtual memory abstracted from the C implementation memory.

The correctness (coupling) invariants from Tab. 1 are specified as 1-state invariants over the data structures of the hypervisor and over the simulated virtual hardware. More precisely, the invariants specific to a single virtual processor are included in the invariant of the implementation data structure of type `Vp` (Fig. 1) and the invariants establishing properties over the VPs altogether (s.t. the invariant *distinct_asids*) are specified in the data structure of type `Guest`. With each VP we associate a set of ghost fields used for maintaining correctness of the SPT algorithm (e.g., maps of allocated and reachable SPTs for this VP).

The properties of the overall system which have to be maintained by software and hardware steps are specified in the so called *hardware interface*. For instance, it specifies for each HP a map $W[i]$ (Sec. 3.4), which contains all walks possibly residing in the HTLB of that processor, and states the invariant *htlb_walks* (Tab. 1). The hardware interface is purely ghost, since it is only used for specification rather than to implement concrete data structures or hardware components. To check that the invariants of the hardware interface are maintained by all possible hardware transitions, we have to explicitly invoke each of them in the MMU thread.

4.1 Specification

The processor state[6] is modeled using the struct type `Processor`.

```
spec(typedef struct _Processor {
    Procx i; // Processor id
    bool v; // flag for virtual
    volatile Asid asid; // processor ASID
    volatile Tlb tlb; // TLB (a map of walks)
    Hardware *h; // pointer to hardware container
    Hwinterface *hwi; // pointer to HW interface
    inv(approves(h, tlb, asid)) // approval by hardware
    inv(!v ==> approves(hwi, tlb, asid)) // approval by HWI
    inv(approves(owner(this), asid)) // thread approval
    inv(v ==> approves(owner(this), tlb)) //thread approval
} Processor;)
```

Fields of the processor which may change only by instruction execution (as e.g. registers) are approved by the running thread. (We mark these sequential fields volatile to allow them being controlled by the 2-state transition invariant of the hardware.) The flag v is used to distinguish between the host and the virtual hardware. For the virtual hardware the TLB of the processor is also approved by the running thread (the steps of the VTLB are always explicitly performed by intercept handlers). For the host hardware, the TLB and the current ASID register are approved by the hardware interface, where we state software dependent properties on these fields (see Fig. 1 for dependencies between data structures used for hardware modeling and implementation of the algorithm).

The data structure `Hardware` encapsulates all processors and defines via 2-state invariants all valid hardware transitions. To ensure that the processor respects this transition relation, it has to approve all processor fields.

```
spec(typedef struct _Hardware {
    Processor *p; // array of processors
    bool v; // flag for virtual
    claim_t cm[Ppfn], cp[Procx]; //claims on processors and memory
    Ppfn gpa2hpa[Ppfn]; // memory translation (for VM)
    volatile Procx i; // index of acting processor
    volatile Action act; // type of action
    volatile Walk w; // TLB walk for the action
    inv(forall(Procx i; claims_obj(cp[i], &p[i]))) // Claims on processors
    // Claims on memory (for VM)
    inv(forall(Ppfn a; gpa2hpa[a] ==> claims_obj(cm[a], page(gpa2hpa, a))))
    inv(p_unch(p) && (!v || m_unch(abs_m(gpa2hpa))) ||
      act == TLB_SET_AD && tlb_setad(p, i, w, old(read_pte(w,gpa2hpa,v)))
        && (!v || m_upd(abs_m(gpa2hpa), w)) ||
      act == CORE_INVLPGA && core_invlpga(p, i, v)
        && (!v || m_unch(abs_m(gpa2hpa))) || ...) // Transition relation
} Hardware;)
```

[6] We expose only the most crucial parts of the data structures necessary to understand our methodology, omitting e.g., valid ASIDs and *CR3* registers here.

```
typedef struct _Spt {
    volatile Pte e[512];
    spec(volatile Pte ge[uint];)
    inv(approves(owner(this), ge))
    inv(sptes_eq_except_a_and_d(e, ge))
} Spt;
```

```
typedef struct vcc(claimable) _Gpt {
    volatile Pte e[512];
    spec(_Hardware *h;)
    inv(approves(h, e))
} Gpt;
```

Listing 1. SPTs and GPTs

The current hardware transition is specified by variables i, act, and w, where i identifies the acting processor, act the action type and w the walk targeted by the action in case of a TLB transition. When we prove simulation for the VMs, we use these variables to choose a certain step we want to simulate. In case of the host hardware these variables allow us to explicitly go over all possible TLB steps in the MMU thread, showing that none of them violate the VCC software invariants.

In the hardware transition system we make a distinction between the steps of the host hardware and the steps of the virtual hardware, having memory framing only if the v bit is set. For the virtual hardware we also require that it claims all memory pages allocated the the VM. We obtain those pages by the map $gpa2hpa$ translating guest physical addresses to host physical addresses (constructed during VM initialization and maintained during memory allocation). An arbitrary memory page of the VM is modeled as a GPT (Listing 1) consisting of guest PTEs (GPTEs) which are approved by `Hardware`.

4.2 Implementation

A SPT (Listing 1) contains an array of SPTEs. Since the A/D bits of a SPTE may be accessed concurrently by hardware, we have to mark the complete entry as volatile. All other bits may only be accessed by the software currently running on the processor. Since thread approval can not be stated bit-wise we have to introduce an approved ghost copy of each SPTE (identical to the original one except for A/D bits).

The SPT algorithm itself consists of a number of intercept handlers. The most crucial ones are considered below.

#PF Intercept. When a #PF is intercepted, the hypervisor walks the GPTs to obtain a set of GPTEs used for the translation of the faulty VA. Every access to a GPT is performed inside an atomic block. During the walk we simulate the respective VTLB steps (initializing a walk, setting A/D bits, extending a walk), which makes the GPT walker the core part of the #PF handler. If the walk extension is unsuccessful (rights violation or present bit not set), we simulate the #PF-signalling step of the VP, inject #PF to the VM and return. After successful walking of GPTs the handler walks the SPTs and finds the first SPTE which is not in-sync with the associated GPTE. The subtree pointed by this SPTE is freed and new subtree (being in-sync with the fetched GPTEs) is allocated and attached to the SPTE. The set of walks $W[i]$ is updated to hold the newly attached walks and to drop the detached ones. Non-dirty terminal SPTEs are marked write protected to propagate a D bit to the VM when it is being set by the HTLB. In case of detaching a subtree we perform a hardware `INVLPGA` to ensure that the HTLB is not sitting on the freed SPTs.

Flush Intercept. When the TLB flush is intercepted, the handler frees all the SPTs of the VP, allocates a fresh top-level SPT (which has all its entries set to non-present), assigns an unused ASID for the VP and simulates the VTLB flush step. If all the ASIDs are already in use, the handler flushes the HTLB and increments the ASID generation of the HP. (Currently we explicitly assume that the ASID generation doesn't overflow.) At this point, all ASIDs which were previously assigned to VPs running on this HP become invalid. The handler then gives the first ASID to the intercepted VP and makes it valid by setting the ASID generation of the VP to the current one of the HP. Set $W[i]$ in the hardware interface is updated to hold only walks sitting on the fresh top-level SPT.

Every time when some VP is scheduled to run we check whether the ASID generation of the VP is equal to the ASID generation of the HP. If this is not the case (i.e., some other VP has increased the ASID generation of the HP), we allocate an unused ASID for the VP and proceed in the same way as in the case of a flush intercept.

`INVLPG` **Intercept.** In case of the `INVLPG` intercept the handler walks down the SPTs for the invalidated address and marks a terminal SPTE non-present. Then it performs a hardware `INVLPGA` on the faulty VA in the ASID of the intercepted VP. The complete walk through the modified terminal SPTE is removed from the set W and the `INVLPG` step of the virtual VP is simulated.

4.3 Verification

We consider verification examples of the code of the #PF handler simulating the step of the VTLB and, of a single HTLB transition performed in the MMU thread.

VTLB Steps Simulation. The VTLB operates on the shared (volatile) memory of the VM and races with other VTLBs and with the running VPs. Hence, the memory of the VM may change arbitrarily in between atomic accesses to it. To simulate a VTLB step corresponding to the operation on the memory of the VM, we have to perform the simulation in the same atomic block where the handler reads/writes GPTs. We also need to have access to the state of the virtual processor and to the transition relation of the virtual hardware. The VP and everything in its ownership domain is thread local, while an instance of `Guest` is shared between all the VPs and is claimed to be closed by a claim gc (i.e., we can not update sequential data of the guest, but can assert its invariant).

As an example of a VLTB step we consider the setting of A/D bits for a top-level walk (performed in a GPT walker before we fetch a top-level GPTE for walk extension):

```
pfn = gpa2hpa(vp−>gwo, guest);
if (pfn == 0) return 0; // non−allocated guest address
gpt = (Gpt ∗)(pfn << 12);
px = compute_idx(vpfn, 4);
while (!cmp_result)
  writes(vp)
  inv(thread_local(vp) && claims(gc, guest) && ...)
{
    atomic(gpt){old_pte = gpt−>e[px];} //fetching GPTE
    unwrap(vp); // opening thread−local object
    atomic(...){ // setting A and D bits
        if (old_pte.p) { // modifying and writing GPTE
            cmp_result = (old_pte == (rw && old_pte.rw)
                ? asm_cmpxchg(&gpt−>e[px], old_pte, SET_AD(old_pte))
                : asm_cmpxchg(&gpt−>e[px], old_pte, SET_A(old_pte)));
            spec(if (cmp_result) { // fixing step parameters
                guest−>g.i = vp−>i;
                guest−>g.act = TLB_SET_AD;
                guest−>g.w = top_level_walk(vp−>gwo, vpfn);
              })
        } else // don't do update if the entry is not present
            cmp_result = 1;
    }
    wrap(vp); // closing thread−local object
}
```

Since the x64 architecture does not provide an instruction performing an atomic read-modify-write operation we use a loop in which we fetch an entry, modify it, and then write it back if the entry has not been changed in between. Writing to a GPTE is done by an interlocked compare-exchange operation. To specify the behaviour of compare-exchange we define a C function `asm_cmpxchg` reflecting the effect of the interlocked operation on the C memory. If the compare-exchange is successful, we simulate the setting of A/D bits by the VTLB.

The invariants of the virtual hardware are checked automatically at the end of the atomic block, ensuring that a selected TLB step is performed accordingly to the transition relation. Since we operate only with one VP, VCC doesn't need to check the invariants of other VPs. The invariants of the hardware interface also are untouched here, because the set of the reachable walks remains unchanged.

MMU Thread. For soundness of the approach we have to emulate the behaviour of a hardware MMU in a C thread. There are two reasons why the two-state invariant of `Hardware` describing the MMU behaviour alone is not enough. First, the only place where VCC checks that the invariant of the concurrent object being modified holds is the atomic block where the writing to the object is done. Moreover, to check this VCC first has to ensure that the address being written belongs to a typed object. In the MMU thread we guarantee that all MMU writes are done to the SPTs of a running VP and these writes do not violate the invariants of SPTs.

The second reason why we need a software MMU thread is the presence of the observer: the invariants of the hardware interface should not restrict the hardware transition system in any sense. Since the transition invariant contains a disjunction of steps, we have to ensure that the invariant of the hardware interface holds for every step from the disjunction. Note, that the invariant of the hardware interface has to be checked not only for MMU steps, but for other hardware steps as well. This check is done in the intercept handlers every time we execute a certain TLB-dependent processor step (e.g., INVLPGA).

The MMU thread consists of a number of atomic actions each performing a single MMU step. As an example, we again consider the setting of A/D bits by the TLB of the hardware processor *hp*.

```
atomic(...) {
    spt = (Spt ∗)(w.pfn << 12);
    px = compute_idx(w.vpfn, w.l);
    assume(hp−>tlb[w] && w.l != 0 && w.asid == hp−>asid
        && hp−>asid > 0 && spt−>e[px].p); // assuming guard
    vp = guest−>hp2vp[hp−>i][hp−>asid]; // get the running VP
    assert(inv(vp)); // asserting invariant of running VP
    begin_update(); // start of update in the block
    spt−>e[px] = (w.l == 1 && w.r[rw] && spt−>e[px].rw)
        ? SET_AD(spt−>e[px])
        : SET_A(spt−>e[px]); // performing a write
    guest−>h.i = hp−>i; // fixing step parameters
    guest−>h.act = TLB_SET_AD;
    guest−>h.w = w;
}
```

With the help of the invariant of the running VP (obtained from the current ASID of the HP), VCC derives that the memory write is done to the SPT owned by that VP and the system invariants are maintained. Note, that the only invariants which might get broken by the HTLB step are the invariant of the hardware interface (if the HTLB adds a walk which is not present in $W[i]$) and the invariant of the SPT itself (if the HTLB modifies other bits than A/D).

5 Conclusion and Future Work

We have demonstrated the verification of a concurrent program dealing with devices using an automatic C code verifier. We have given a general methodology for verification of virtual device implementations, specified TLB virtualization with SPTs and formally verified a SPT algorithm.

The implementation of the SPT algorithm contains ca. 700 lines of C code (including initialization of data structures) and ca. 4K lines of the annotations which include function contracts, loop invariants, data invariants, ghost code, and (proof) assertions. Roughly a third of annotations comprise function and block contracts and another third is ghost code for maintaining ghost fields, showing simulation, and running MMU thread (which is purely ghost). The overall proof time is ca. 18 hours on one core of 2GHz Intel Core 2 Duo machine. The estimated person effort is 1.5 person-years, including VCC learning period.

There are two possible directions of future work. The first one is to integrate the proof and the specification to a prototypical hypervisor being developed and verified at the Saarland University. In particular, this requires adapting the proof to be done on top of the kernel layer of the hypervisor, rather than on top of the real hardware (the support for this scenario is already included in our models). The second direction is to verify a more sophisticated version of the algorithm, which uses write-protection of GPTs and sharing of SPTs.

References

1. Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.14 edn. (September 2007)
2. Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In: Formal Methods in Computer Aided Design (FMCAD) 2010, pp. 267–270. IEEE, Lugano (2010)
3. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
4. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
5. Intel Corporation: TLBs, Paging-Structure Caches, and Their Invalidation (April 2007)
6. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 207–220. ACM, New York (2009)
7. Tews, H., Weber, T., Völp, M., Poll, E., van Eekelen, M., van Rossum, P.: Nova micro–hypervisor verification. Tech. Rep. ICIS–R08012. Radboud University Nijmegen (May 2008)

Formalization and Analysis of Real-Time Requirements: A Feasibility Study at BOSCH

Amalinda Post[1] and Jochen Hoenicke[2]

[1] Robert Bosch GmbH, Stuttgart, Germany
amalinda.post@de.bosch.com
[2] University of Freiburg, Germany
hoenicke@informatik.uni-freiburg.de

Abstract. In this paper, we evaluate a tool chain to algorithmically analyze real-time requirements. According to this tool chain, one formalizes the requirements in a natural-language pattern system. The requirements can then be automatically compiled into formulas in a real-time logic. The formulas can be checked automatically for properties whose violation indicates an error in the requirements specification (the properties considered are: consistency, rt-consistency, vacuity). We report on a feasibility study in the context of several automotive projects at Bosch. The results of the study indicate that the effort for the formalization of real-time requirements is acceptable; the analysis algorithms are computationally feasible; the benefit (the detection of specification errors resp. the formal guarantee of their absence) seems significant.

1 Introduction

According to common industrial practice, requirements are specified in natural language and checked for errors manually, e. g., by peer reviews [16]. The shortcomings of this tool chain are well-known: the disambiguation of the (natural language) requirements is done by component specialists (instead of system specialists) during implementation and testing; both, the cost and the error detection rate of the manual checks do not scale well with the number of requirements, which each affect another and cannot be analyzed in isolation [3]. Further, a review can detect errors but never guarantee their absence.

A tool chain for the formalization of requirements and the (subsequently possible) formal, automatic analysis of requirements opens the perspective of eliminating the above shortcomings. Much research has been invested recently in language and tool support for both, formalization and analysis, e. g., [14,6,7,18,13,12]. The question whether such a tool chain is feasible in practice can not be decided by a principled argument that applies uniformly to all practical settings; we need a number of feasibility studies which address the question on a case-by-case basis. This paper presents such a study, for a special case of behavioral requirements, namely real-time requirements, in the context of several automotive projects at Bosch. We call a requirement *real-time requirement* if it contains an explicit timing bound, e. g., "If IRTest is set then the infrared lamps are turned on *after at most 10 s*."

We believe that real-time requirements are a good 'first target' for a feasibility study. Their formulation tends to be concrete; i. e., they are more amenable to formalization than other requirements. Real-time requirements are notoriously hard to get right, and they appear in projects for safety-critical systems; i. e., the extra need for quality assurance efforts is widely accepted. (The same reasons gave the incentive for previous work on real-time requirements [14,13,12]).

This paper contributes a first comprehensive evaluation of a tool chain for the algorithmic analysis of real-time requirements in a particular industrial setting. The tool chain contains, in addition to the algorithms proposed in [13,12], also a user-friendly input language which is indispensable in an industrial setting. In particular, we combine a specialized specification language based on a system of natural-language patterns [9,14] with analysis tools for requirements in a real-time logic [13,12]. To allow that we developed a compiler from the specification language into the real-time logic used in [13,12]; see Figure 1.

We perform a feasibility study in the context of several automotive projects at Bosch and evaluate the tool chain in terms of the human effort required for the formalization, the performance of the tools, and the outcome of the application of the tool chain to each of the examples in the case study. The overall result of the evaluation indicates that the tool chain is feasible and worthwhile. We will now recount the results of the evaluation in greater detail.

Results of the Evaluation. As expected, the effort for the formalization of real-time requirements was heavy. Two to three minutes per requirement in average seems still acceptable, however (in contrast with analysis), the formalization of requirements is done one by one; i. e., it scales linearly in the number of requirements. We chose the setting for our study where we start with already existing sets of documented informal requirements. This separation between requirements elicitation and formalization allows us to measure the effort spent for the formalization. The by far largest chunk of the measured effort goes into understanding the requirements. We thus obtain a safe approximation of the effort that is spent in the (preferable) setting where the formalization is interleaved with the elicitation.

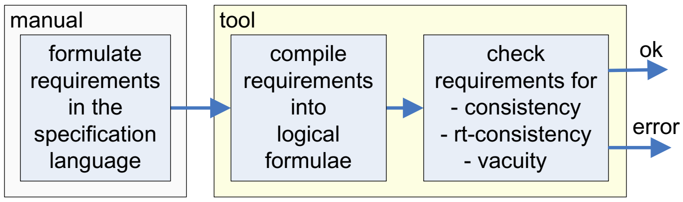

manual
formulate requirements in the specification language
tool
compile requirements into logical formulae
check requirements for
- consistency
- rt-consistency
- vacuity
ok
error

Fig. 1. The tool chain evaluated in the feasibility study. The compiler from the specification language to the real-time logic interfaces the manual formalization and the automatic analysis; see also Figure 2.

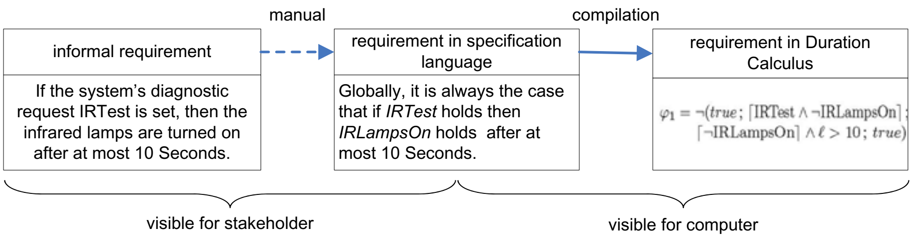

manual
compilation
informal requirement
If the system's diagnostic request IRTest is set, then the infrared lamps are turned on after at most 10 Seconds.
requirement in specification language
Globally, it is always the case that if *IRTest* holds then *IRLampsOn* holds after at most 10 Seconds.
requirement in Duration Calculus
$\varphi_1 = \neg(true; \lceil \mathrm{IRTest} \wedge \neg \mathrm{IRLampsOn} \rceil ; \lceil \neg \mathrm{IRLampsOn} \rceil \wedge \ell > 10; true)$
visible for stakeholder
visible for computer

Fig. 2. In the tool chain, a specification language for real-time requirements is used as an intermediate step between the informal part and the formalism used as input for the analysis tools (the transition from the specification language to the input format is automatic, i. e., done by a compiler)

We already knew that the computational complexity of the analysis algorithms is high and that the tools still need to be optimized [13,12]. The situation is comparable to model checking in that one cannot expect the tools to scale uniformly, and the tools need to be specialized to the application domain. The performance of the tools in our study (which ranges from several seconds to more than an hour) is too irregular for an on line use, e. g., interleaved with elicitation; for batch processing, as with our tool chain, the execution times on the examples in our study are more than acceptable.

The benefit of applying the tool chain seems significant. The detection of several specification errors in examples that had undergone extensive reviews is a benefit of obvious practical value. That is, the tool chain has allowed us to find errors that had escaped the reviews. Each error was detected because of the violation of one of three correctness properties. If we had a larger set of correctness properties that our tool chain could use, we might detect further errors still hidden in the requirements specification. This opens an avenue for further research.

The other kind of benefit, i. e., the formal guarantee of the absence of a particular kind of error, is of a purely conceptual value. It is interesting to note, however, that engineers at Bosch are quite keen on this functionality of the tool chain. This is noteworthy since engineers are reputed to be pragmatic. Perhaps mathematical certitude is an innate universal need, after all.

2 The Tool Chain

As depicted in Figure 1, the first step of the tool chain is manual and the second is fully automatic. In the first step, the requirements engineer formalizes real-time requirements in the specification language. The second step is a call to a tool (with, as front end, a compiler from the specification language to the proper input format of the analysis tools). We now briefly present the specification language, the different properties checked by each of the analysis tools, and the analysis tools themselves.

In this paper, we rely on the results which state the correctness of the analysis tools which we use in our case study; for completeness, we will explain the properties checked by the analysis tools but we must refer to [14,13,12] for further details about the foundation of the algorithms used in the tools.

2.1 The Specification Language

The specification language is depicted in Table 1. It is a restricted English grammar based on the specification pattern system (SPS) given by Konrad and Cheng [9].

Every pattern consists of non-literal terminals P, Q, c and literal terminals. For example, in the *bnd response* pattern "it is always the case that if P holds, then S holds after at most c time unit(s)", P, S, and c are (the only) non-literal terminals. The non-literal terminals P and S denote boolean propositional formulae that capture properties of the system. The non-literal terminal c is instantiated with constants. In the example of a requirement (in the specification language) given in Figure 2, the pattern is instantiated by setting P to *IRTest*, S to *IRLampsOn*, and c to 10.

The specification language is geared toward a person who is not formally minded [14]. The use of the specification language in our tool chain is possible only thanks to the compilation from the specification language into the minimalistic formalism used for the input of the analysis tools; see Figure 2. The stakeholder only needs to care about the formulation of the requirements in the specification language; their formulation in the real-time logic (the input format of the analysis tools) is irrelevant for the stakeholder and only relevant for the analysis tool.

2.2 Translation of the SPS to Duration Calculus

Konrad and Cheng provide a translation of their SPS to the logics TCTL (Timed Computation Tree Logic), RTGIL (Real-Time Graphical Interval Logic) and MTL (Metric Temporal Logic). In this work we translate the SPS to the Duration Calculus fragment defined in [8]. Table 2 depicts our translation. These formulas are further translated into Phase Event Automata (PEA) [8], on which the consistency properties are checked.

Note that for the *eventually pattern* and the *response pattern* we need for the scopes *Globally, After Q, After Q until R* the translation uses the Duration Calculus operator $\rhd$ introduced by Skakkebæk [15]. For these six instances the algorithms to check rt-consistency and vacuity cannot be directly applied, as the algorithm to calculate a PEA representing a requirement is not defined for requirements with the $\rhd$ operator. We circumvent this problem in our tool in having defined the corresponding PEAs by hand.

2.3 The Correctness Properties

In the tool chain we check requirements for three properties: *inconsistency, rt-inconsistency* and *vacuity*.

Table 1. Restricted English grammar based on the grammar given by Konrad and Cheng in [9]

Start	1: property	::= *scope specification* .
Scope	2: scope	::= Globally | Before R | After Q | Between Q and R | After Q until R
General	3: specification	::= *qualitative* | *real-time* | *invariant*
qualit.	4: qualitative	::= *absence* | *universality* | *existence* | *bnd existence* | *precedence* | *response*
	5: absence	::= it is never the case that P holds
	6: universality	::= it is always the case that P holds
	7: existence	::= P eventually holds
	8: bnd. exist.	::= transitions to states in which P holds occur at most twice
	9: precedence	::= it is always the case that if P holds, then S previously held
	10: response	::= it is always the case that if P holds then S eventually holds
real-time	11: real-time	::= *min duration* | *max duration* | *bnd recurrence* | *bnd response* | *bnd invariance*
	12: min dur.	::= it is always the case that once P becomes satisfied, it holds for at least c time unit(s)
	13: max dur.	::= it is always the case that once P becomes satisfied, it holds for at most c time unit(s)
	14: bnd recur.	::= it is always the case that P holds at least every c time unit(s)
	15: bnd resp.	::= it is always the case that if P holds, then S holds after at most c time unit(s)
	16: bnd inv.	::= it is always the case that if P holds, then S holds for at least c time unit(s)
invariant	17: invariant	::= it is always the case that if P holds, then S holds as well

We say that a set of requirements φ is *inconsistent* if there exists no system satisfying φ, e. g., it exists no system satisfying both "Req_1: Once *IRTest* holds it holds for at least 5 Seconds." and "Req_2: Once *IRTest* holds it holds for at most 3 Seconds."

The check for *rt-inconsistency* analyzes whether timing bounds of real-time requirements may be in conflict. The formal definition of rt-inconsistency is given in [13], e. g., the following two requirements are *consistent* but not *rt-consistent*: "Req_3: Globally, it is always the case that if *IRTest* holds, then *IRLamps* holds after at most 10 seconds", "Req_4: Globally, it is always the case that if *IRTest* holds, then ¬*IRLamps* holds for at least 6 seconds". Say the observable *IRTest* holds from time point 4 on for 6 seconds (as depicted in Figure 3). Then Req_3 requires that *IRLamps* appears not later than $t = 14$. At the same time Req_4 requires that *IRLamps* does not hold until at least $t = 16$—a conflict. Formally, a set of requirements is *rt-inconsistent* if there is a a finite trace satisfying all requirements that cannot be extended to an infinite trace.

Table 2. Translation of the SPS into Duration Calculus (Excerpt)

Scope	Pattern	Duration Calculus
Globally	it is never the case that P holds	$\neg(true; \lceil P\rceil;\ true)$
Before R		$\neg(\lceil\neg R\rceil; \lceil\neg R \land P\rceil; \lceil\neg R\rceil;\ true)$
After Q		$\neg(true; \lceil Q\rceil; true; \lceil P\rceil;\ true)$
Between Q and R		$\neg(true; \lceil Q \land \neg R\rceil; \lceil\neg R\rceil; \lceil P \land \neg R\rceil; \lceil\neg R\rceil; \lceil R\rceil;\ true)$
After Q until R		$\neg(true; \lceil Q \land \neg R\rceil; \lceil\neg R\rceil; \lceil P \land \neg R\rceil;\ true)$
Globally	P eventually holds	$(\neg(\lceil\neg P\rceil)) \rhd true$
Before R		$\neg(\lceil\neg R \land \neg P\rceil; \lceil R\rceil;\ true)$
After Q		$(\neg(true; \lceil Q \land \neg P\rceil; \lceil\neg P\rceil)) \rhd true$
Between Q and R		$\neg(true; \lceil Q \land \neg R\rceil; \lceil\neg P \land \neg R\rceil; \lceil R\rceil;\ true)$
After Q until R		$\neg(true; \lceil Q \land \neg R\rceil; \lceil\neg P \land \neg R\rceil; \lceil R\rceil;\ true) \land (\neg(true; \lceil Q \land \neg P \land \neg R\rceil; \lceil\neg P \land \neg R\rceil)) \rhd true$
Globally	it is always the case that if P holds then S eventually holds	$(\neg(true; \lceil P \land \neg S\rceil; \lceil\neg S\rceil)) \rhd true$
Before R		$\neg(\lceil\neg R\rceil; \lceil P \land \neg S \land \neg R\rceil; \lceil\neg S \land \neg R\rceil; \lceil R\rceil;\ true)$
After Q		$(\neg(true; \lceil Q\rceil; true; \lceil P \land \neg S\rceil; \lceil\neg S\rceil)) \rhd true$
Between Q and R		$\neg(true; \lceil Q \land \neg R\rceil; \lceil\neg R\rceil; \lceil P \land \neg R \land \neg S\rceil; \lceil\neg R \land \neg S\rceil; \lceil R\rceil;\ true)$
After Q until R		$(\neg(true; \lceil Q\land\neg R\rceil; \lceil\neg R\rceil; \lceil P\land\neg S\land\neg R\rceil; \lceil\neg S\land\neg R\rceil)) \rhd true \land (\neg(true; \lceil Q\rceil; true; \lceil P \land \neg S\rceil; \lceil\neg S\rceil)) \rhd true$
Globally	it is always the case that if P holds then S holds after at most c time units	$\neg(true; \lceil P \land \neg S\rceil; \lceil\neg S\rceil \land \ell > c;\ true)$
Before R		$\neg(\lceil\neg R\rceil; \lceil\neg R \land P \land \neg S\rceil; \lceil\neg R \land \neg S\rceil \land \ell > c;\ true)$
After Q		$\neg(true; \lceil Q\rceil; true; \lceil P \land \neg S\rceil; \lceil\neg S\rceil \land \ell > c;\ true)$
Between Q and R		$\neg(true; \lceil Q\land\neg R\rceil; \lceil\neg R\rceil; \lceil P\land\neg R\land\neg S\rceil; \lceil\neg S\land\neg R\rceil\land\ell > c; \lceil\neg R\rceil; \lceil R\rceil;\ true)$
After Q until R		$\neg(true; \lceil Q\land\neg R\rceil; \lceil\neg R\rceil; \lceil P\land\neg R\land\neg S\rceil; \lceil\neg S\land\neg R\rceil\land\ell > c;\ true)$

The check for *vacuity* checks whether there is a requirement in the set that is only vacuously satisfied in the context of the set. The formal definition is given in [12], e. g., the following requirements are *consistent* and *rt-consistent* but *vacuous*: "Req_5 : Globally, it is always the case that if *IRTest* holds then *IRLamps* holds after at most 10 seconds", "Req_6: Globally, it is never the case that *IRTest* holds". In every system satisfying both requirements the observable *IRTest* never holds (according to Req_6), i. e., the precondition of Req_5 never holds. Thus, in the context of the set of requirements Req_5 is only *vacuously satisfied*. We assume that a requirements engineer specifies only requirements with behavior that shall be visible in a system, thus we assume that there is an error in the requirements and call Req_5 *vacuous* with the set of requirements.

To formally define vacuity, a purely syntactical characterization of *simpler* requirements is needed. In our case a requirement is of the form $\neg(\varphi_1; \ldots; \varphi_n; true)$ and simpler requirements are those, where some trailing phases $\varphi_i; \ldots; \varphi_n$ are omitted. Then a requirement φ is *vacuous* in the context of a set of requirement, if there is a simpler requirement that is equivalent in the context. For example,

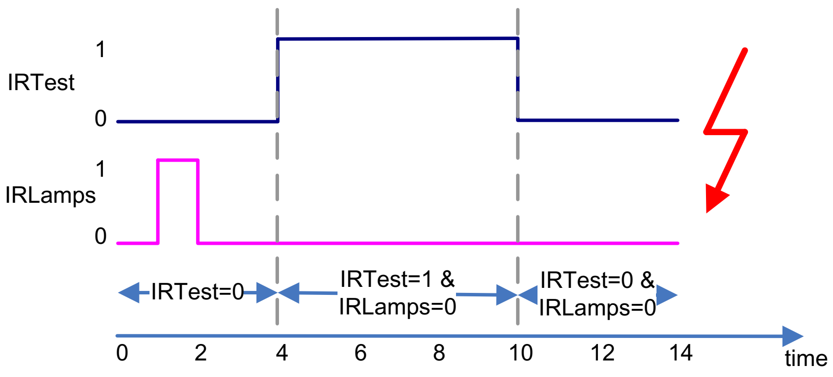


Fig. 3. Req_3 and Req_4 are rt-inconsistent

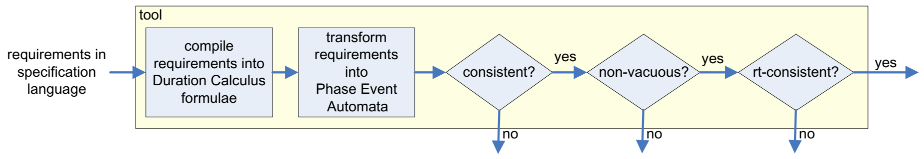

requirements in specification language
tool
compile requirements into Duration Calculus formulae
transform requirements into Phase Event Automata
consistent?
yes
no
non-vacuous?
yes
no
rt-consistent?
yes
no

Fig. 4. The prototype tool compiles requirements in specification language to Duration Calculus formulae; it then starts the analysis tools, i. e., it transforms the logical formulae into a Phase Event Automaton and then checks for consistency, vacuity, and rt-consistency

if in some context, the requirement "after Q it is never the case that P holds", $\neg(true; \lceil Q \rceil; true \lceil P \rceil; true)$, is equivalent to "it is never the case that Q holds", $\neg(true; \lceil Q \rceil; true)$, we say that the first requirement is vacuous in that context.

2.4 The Tool

We have assembled a prototype tool that parses requirements formalized in the specification language and automatically transforms them into formulae in a real-time logic (the Duration Calculus); it then analyzes the formulae to check the formalized requirements for *consistency*, *rt-consistency* and *vacuity*; see Figure 4. The tool is written in Java and bases on the PEA-toolkit developed by [11] and the model checker UPPAAL [2]. Every algorithm is sound and complete (i. e., for every input data the decision procedure returns a correct answer). If the set of requirements is rt-inconsistent, the tool returns a counterexample (a possible behavior that leads to a timing conflict). If the set of requirements is incongruous then the tool returns the requirement that is incongruous in the set.

3 Planning of the Feasibility Study

3.1 Study Goals and Questions

In order to assess the practical relevance of the tool chain in the automotive context, two main questions must be addressed. First, what is the benefit of the tool chain? Second, what are the costs of applying the tool chain?

A preliminary question is whether requirements engineers and software developers are in principle willing to use the specification language for requirements. We addressed this question in an informal inquiry, before starting the actual feasibility study which addresses the two questions above. We asked requirements engineers at Bosch to take some of their behavioral requirements and reformulate them in the specification language. We showed requirements formalized in the specification language to software developers at Bosch and asked them to explain their meaning to us. The reaction was only positive; i. e., the specification language seems easy to use, both for writing and reading. In the context of this inquiry, the request for tool support was constantly repeated.

Before developing a tool that is fit for an industrial use, we decided to first develop a prototype tool and start with evaluating the two questions above in a feasibility study on requirements of different Bosch projects. To summarize, we identified the following two items for evaluation in the study.

(Benefit). Is the tool chain useful in terms of quality assurance for requirements, resp., does it help to identify errors that were not detected in a manual review? Does it support a user in resolving the errors?

(Cost). What effort (measured in time) is needed to formulate requirements in the specification language? Are the analysis algorithms computationally feasible for the examples in the study?

3.2 Selection of the Sample

In the first step we selected requirements documents from different Bosch projects of the automotive domain. To get a representative sampling, we decided to apply stratified sampling over the automotive application domains *driving assistance, engine controlling, car multimedia, catalytic converter development* and *power train* development. We then used convenience sampling to select a project out of every stratum.

Each project had several requirements documents, some consisting of more than 100 pages. In order to get a representative sample of requirements we asked the corresponding requirements engineers to give us 1 to 4 sets of requirements (representative for their domain and containing behavioral requirements and real-time requirements), each specifying a system component. This way we obtained sixteen sets of requirements.

3.3 Feasibility Study Design

In the first step we formulated the requirements in the specification language. Every set of requirements was then again reviewed with feedback from the responsible requirement engineers, and, if needed, changed until we agreed that the meaning of the informal requirements was accurately represented in the requirements formulated in the specification language. For one project we let the requirements engineer directly translate the requirements in SPS, in this case we were just available as coach. We then used the tool to automatically transform the requirements into Duration Calculus formulae. We checked the requirements with the help of our tool for *consistency, rt-consistency* and *vacuity* on a PC Windows XP system with 2 GHz Intel Core 2 Duo processor and 1 GB RAM, whereas only one core was used. If the tool detected an error we searched the reason for the error, fixed it and then checked the set again (until the set was consistent, rt-consistent and non-vacuous). We measured the execution time as CPU-time needed to parse the requirements, transform them to Duration Calculus formulae and then do the respective check.

4 Analysis of the Results

4.1 Benefit

Benefit. Table 3 depicts the validation results for each component. The specifics of the components are not relevant; hence we do not present them and just number the examples from 1 to 16 (first column). The second column refers to the size of the input in the number of requirements. Columns 3 to 6 refer to the outcome of the consistency/vacuity/rt-consistency check.

As Table 3 shows, every component (except Component 13 and 14) was consistent. We guess that this indicates that the manual review process successfully detected any inconsistencies. For Component 13 and 14 we got an out-of-memory error thus we could not determine the consistency.

Regarding vacuity, the automatic validation could guarantee the absence of vacuities for 12 components, in one component it detected an error. In Component 8 the precondition of the following requirement was never satisfied *"If* $accelerationPedal = 0$ *and brakePedalActivated then regeneration holds after at most 1 time unit."* Debugging the requirements we found out that another requirement was ambiguously specified as *"The value range of the acceleration pedal is between 0 and 100."* This second requirements was misinterpreted as "$0 < accelerationPedal < 100$" instead of "$0 \leq accelerationPedal \leq 100$". We resolved the ambiguity and changed the requirement to *"The value range of the acceleration pedal is between 0 and 100 where the endpoints of the interval are included"*. The needed change was only a minor change, but the analysis helped to discover an ambiguous requirement. Thus, the check for vacuity was beneficial. Note that only the biggest component contained an vacuity. We see two possible reasons for that. First, it might be that in practice vacuity only rarely occurs. Second, it might be that vacuities occurred, but they were already detected in the manual reviews and subsequently resolved. In the belief of the requirements engineers at Bosch, vacuity occurs in practice. Thus, we believe that the vacuities were resolved in the earlier steps. This would also explain, why the vacuity was detected in the biggest component—large components are

Table 3. Checking consistency, rt-consistency and vacuity for existing examples of sets of real-time requirements for software components in automotive projects at Bosch using a prototype implementation (Fig. 4)

	$\#_{req}$	consistent?	non-vacuous?	rt-consistent?
1	9	yes	yes	no
2	10	yes	n/a	no
3	10	yes	yes	no
4	12	yes	yes	yes
5	13	yes	yes	yes
6	17	yes	yes	yes
7	17	yes	yes	no
8	18	yes	yes	no
9	27	yes	yes	yes
10	27	yes	yes	yes
11	29	yes	yes	no
12	40	yes	yes	no
13	48	n/a	n/a	n/a
14	58	n/a	n/a	n/a
15	81	yes	no	yes
16	81	yes	yes	yes

much more difficult to review for humans, as it gets difficult to keep the interdependencies in mind.

For three components our algorithm returned with an out-of-memory error. Note that the space of solution tends to explode, if there are not many interdependencies between the requirements in the set. If there are many interdependencies, then the space of solution gets reduced. Thus, the instances that are difficult to check for the tool, are often quite easy to review for a human—and vice versa: the instances with many interdependencies, which are more difficult to grasp for a human, get easy to check for the tool. Thus, for this property it seems that the automatic and the manual analysis might well complement each other.

Most errors were discovered by the rt-consistency-check: seven out of 16 components were in fact rt-inconsistent, i. e., for Components 1, 2, 3, 7, 8, 11 and 12, the check identified flaws in the requirement specification that needed to be repaired. Major changes were needed to correct these requirements. e. g., for Component 3, two of the existing requirements were deleted, five were changed, and seven new requirements were added.

Fixing the errors was an iterative process, with up to 10 iterations. In every iteration we thought to have fixed the problem, but then the check again found an rt-inconsistency. The output-interpretation then helped us to identify the reason of the errors. Thus, although rt-inconsistencies seem to appear frequently in requirements specifications this property seems to be difficult to detect for humans. The benefit of this check is thus very high.

For Component 4, 5, 6, 9, 10 and 15 the tool chain assured the consistency, non-vacuity, and rt-consistency of the requirements. The tool chain helped us to assure the quality of six components, and to detect 8 errors in the requirements that were not known before. Three of the errors were even found in the smallest sets of requirements. We thus think that the tool chain is even beneficial for smaller sets of requirements.

Costs. To evaluate whether the benefit of applying our tool chain justifies its costs, we measure the time needed to formalize the requirements and second the execution time needed by our tool (i. e., the time the requirements engineer needs to wait for the results).

To express the informal requirements in the SPS we needed in average about 2–3 minutes per requirement, i. e., for Component 1 to 9 we needed 26 Min, 30 Min, 28 Min, 35 Min, 25 Min, 28 Min, 32 Min, 38 Min, 60 Min, and 2 h 50 Min for Component 16. Most of the time was needed to understand the meaning of the informal requirement, the reformulation itself was then quickly done. However even two minutes per requirement may scale to a considerable amount of time as in the automotive domain there are often thousands of requirements for one product. However, if the tool chain is integrated within the development processes (i.e, the requirements are directly formulated in the SPS when developing the requirements) then these costs could be omitted.

Secondly, we evaluate the execution time of the tool. If the requirements engineer needs to wait a long time before getting the validation results this is costly. As he is working on other topics in the meantime, he will need some time to familiarize himself again with the requirements, and it will need more time to debug the requirements. In practice, the requirements engineer needs to wait for the results of a manual review for some days. Thus, our tool chain has to compete with that time slot. The execution time for the checks is considerably smaller. The longest execution time took 1 h 32 Min—a big improvement. Thus, with respect to the waiting time, applying the tool chain might even decrease the costs of requirements engineering, as the requirements engineer can validate a set of requirements directly when specifying the requirements.

Over all, it seems that the tool chain is very beneficial and the costs of applying the tool chain are reasonable, they might even slightly decrease. However, there is one restriction: we suspect that for big sets of requirements the space of solutions grows explosively. For Component 13 and 14, our checks returned with an Out-of-Memory-Error. Further optimizations are needed to develop efficient algorithms, still the algorithms have to handle the state explosion problem [19,13]. We think that the tool chain is cost-effective when validating component requirements or sets of component requirements, but probably not suited to validate the set of all system requirements at once. Further studies on bigger sets of requirements are needed to confirm or refute that belief.

4.2 Observations

Usability. In an initial survey we had asked requirements engineers of Bosch to apply the SPS on requirements. The results indicated that they thought the

SPS easy to apply and easy to learn. This was confirmed in our feasibility study: initially, the training curve was steep. For the first requirements the requirements engineer needed up to 10 minutes per requirement to express the requirement in SPS, i. e., to compare the requirement with the available patterns. But once he had used a pattern at least once the needed time considerably decreased to 1–3 minutes. Further, all requirements engineers could directly explain the meaning of a requirement in SPS. Only the use of the different scopes needed some explanation. Thus, we think that the SPS as input language is suited for the whole development team, even without much training.

In contrast, we believe that to interpret the output of the checks a more extensive training is needed. For a given set of requirements we return the check result, and if the set is rt-inconsistent a run to the rt-inconsistency, and if it is vacuous the set of requirements that are only vacuously satisfied. Without training, the engineers could interpret the check results, but they had some problems in interpreting the runs. But without the runs, the requirements are very difficult to debug. Thus, we think that the tool chain allows that many developers specify requirements, and they can do so directly in SPS. But we recommend that only the requirements engineer checks the requirements for consistency, rt-consistency and non-vacuity. This way, only the requirements engineer needs to be trained in interpreting the output of the checks.

For some requirements it was difficult to decide, whether the requirement should be formalized with an "invariant pattern" or with a "(bounded) response pattern", e. g., the requirement *"If the system is in error-mode then AssistFunction has to be deactivated"* might be formalized as *"Globally, it is always the case that if* $errorMode$ *holds then* $\neg AssistFunctionActive$ *holds as well"*, expressing the desired invariant relation between the two variables. However, on a deeper abstraction level, that may not be quite true. Say the system state is calculated in one software function, and $AssistFunction$ is implemented in another function. Both functions are called in the same task, and $AssistFunction$ checks the system state when being called. Then the formalization *"Globally, if* $errorMode$ *holds, then* $\neg AssistFunction$ *holds after at most 10 ms."* would be more appropriate. The natural language requirement may be the same on every abstraction level, although its meaning changes depending on the context. In contrast the formalized requirements must change depending on the abstraction levels—they make the change of the semantics explicit.

Specification Language. We further noticed that if a requirements engineer specified a system component in SPS, then the requirements in SPS tended to contain more information than the ones in natural language, e. g., the requirement *"If the locally measured voltage is not available for Local Voltage Usage (InternVoltageError), the system voltage value as received from the bus shall be used."* was expressed in SPS as *"Globally, it is always the case that if* $\neg BusOff \wedge InternVoltageError$ *holds then* $VoltageValue' = VoltageValueFromBus$ *eventually holds"*. The requirements engineer used the implicit knowledge, that the voltage value received from the bus is only received if the bus is not off. Thus, it seems that the SPS is a way to make implicit knowledge more explicit.

Nearly all requirements in the case study specified invariant or future behavior. The precedence pattern was only used once.Further, about two thirds of the requirements were formalized using the invariant pattern(*it is always the case that if P holds then S holds as well*) or (bounded) response pattern (*it is always the case that if P holds then S eventually holds,* resp., *then S holds after at most C time units.* The other patterns were only rarely used. We believe that this is the case, as for the specification of the behavioral requirements the systems were seen as blackbox. Thus, the resulting requirements mostly relate input and output variables to each other.

What we found a bit surprising was that there was no need to express uncertainty or prioritization in the specification language. When asking the requirements engineers, they said that all requirements needed to be implemented, there are no "nice-to-have" requirements. The only prioritization needed is the one to decide what requirement has to be implemented for what release—but this question was managed with the help of requirements attributes. Furthermore, although there is a high degree of uncertainty in the requirements, this uncertainty is hidden within application parameters, e. g., say *AssistFunction* shall only be active if the vehicle speed is above a certain threshold, however the value of that threshold is not yet known. Then, the requirements engineers invent an application parameter and specify the requirement like, e. g., *"if velocity $\leq$ threshold holds then $\neg$AssistFunctionActive holds as well".* This way, uncertainty is resolved using application parameters.

Change Requests. We detected three items to tailor the specification language by Konrad and Cheng to the automotive domain. First, the requirements engineers asked for a new pattern, specifying invariant behavior. They wanted to express requirements like *if P holds then S holds as well.* In the specification language given by [9] this behavior can be expressed using the *absencepattern*, i. e., via *"it is never the case that $P \wedge \neg S$ holds".* However, the requirements engineers thought it to be not intuitive, to specify invariant behavior via the absence pattern. We thus extended the specification language with the further pattern, to improve the intuitiveness. Second, some requirements engineers asked us to omit the "it is always the case that" in the *precedence, response, min duration, max duration, bnd recurrence-, bnd response* and *bnd invariance* pattern. They noted that this part of the pattern sounded strange in combination with the *Globally* scope. Third, we needed one further pattern. In the error handler concept, errors need to be qualified (i. e., they need to be detected during some time) before they are stored in the error memory. To express such a behavior we needed one further pattern, *"if P holds for at least c time units then S holds after at most c time units."*.

5 Threats to Validity

In this section, we analyze threats to validity defined in Wohlin [17]. Note that the results of the study are only valid in the given context, we do not aim to make any generalizations. Threats to validity concerning the suitability of the specification language are discussed in [14].

5.1 Construct Validity

Expectancy Effect. Expectations of an evaluator toward the outcome can affect a study. We formulated the informal requirements in the specification language. However, the reliability analysis in [14] suggests that applying the specification language is sufficiently independent of the evaluator. Furthermore, the requirements were automatically analyzed by a tool, thus the number of errors is objectively measured.

Inadequate Preoperational Explication of Constructs. This threat arises if the measures are not well defined. In order to minimize that threat we discussed with experts whether our properties *inconsistency, rt-inconsistency* and *vacuity* represent their concept of erroneous behavioral requirements. The discussion indicated that the properties seem to capture erroneous requirements. Furthermore, we formally (i. e., unambiguously) defined the properties. We obtained a safe approximation of the effort that is spent for the formalization and the check in separating elicitation, formalization, and analysis, i. e., in measuring the effort to formulate an informal requirement in specification language (independent of the elicitation effort) and the execution time of the analysis tools.

5.2 External Validity

Sampling validity. This threat arises if the sample is not representative for the requirements. The defects in the requirements specifications might over or under represent the defects in the rest of the corporate requirements specifications. In order to minimize this threat we used the selection procedure described in Section 3.2. A limitation of the feasibility study is that we only used requirements of Bosch projects. Thus we cannot extend our results to the whole automotive domain.

6 Conclusion

The contribution of this paper is twofold. First, we showed how to build-up a tool chain that combines a user-friendly input language (based on a SPS) with an automatic check for consistency, rt-consistency and non-vacuity as proposed in [13,12] for requirements specified in the real-time logic Duration Calculus. Second, we have evaluated the tool chain to algorithmically analyze real-time requirements, in the context of several automotive projects at Bosch. In particular, compared to the case studies in [13,12] we extended the number of samples to 16 samples from six samples in [13] and eleven samples in [12]. The results of the study indicate that the effort for the formalization of real-time requirements is acceptable, that the analysis algorithms are computationally feasible, and that the benefit (the detection of specification errors resp. the formal guarantee of their absence) seems significant. This is encouraging. However, before we can turn the tool chain into a technology that is apt for industrial use, we need to solve a number of research and engineering problems related to scalability and usability. Another avenue for research is to identify more meta-requirements that can be formalized and automatically analyzed, in addition to the three we considered in this paper. The more meta-requirements we have, the more errors in our requirements specifications we will automatically detect (at first), and the more *(mathematically founded)* trust in our requirements specification we will gain (at last).

References

1. Abrial, J.-R.: Formal methods in industry: achievements, problems, future. In: ICSE, pp. 761–768 (2006)
2. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. Dahlstedt, A.G., Persson, A.: Requirements interdependencies - moulding the state of research into a research agenda. In: REFSQ, pp. 71–80 (2003)
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420. ACM, New York (1999)
5. Han, B., Gates, D., Levin, L.: From language to time: A temporal expression anchorer. In: TIME, pp. 196–203 (June 2006)
6. Heimdahl, M.P.E., Leveson, N.G.: Completeness and consistency analysis of state-based requirements. IEEE Trans. on SW Engineering, 3–14 (1995)
7. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. SW Eng. and Meth. 5(3), 231–261 (1996)
8. Hoenicke, J.: Combination of Processes, Data, and Time. PhD thesis, University of Oldenburg (July 2006)
9. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In:ICSE 2005: Proc. 27th Int. Conf. Softw. Eng., pp. 372–381. ACM, New York (2005)
10. Kuhn, T.: Acerules: Executing Rules in Controlled Natural Language. In: Marchiori, M., Pan, J.Z., de Sainte Marie, C. (eds.) RR 2007. LNCS, vol. 4524, pp. 299–308. Springer, Heidelberg (2007)
11. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: a practical approach. Formal Asp. Comput. 20(4-5), 481–505 (2008)
12. Post, A., Hoenicke, J., Podelski, A.: Vacuous of real-time requirements. In: RE 2011, pp. 153–162. IEEE (2011)
13. Post, A., Hoenicke, J., Podelski, A.: rt-inconsistency: A New Property for Real-Time Requirements. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 34–49. Springer, Heidelberg (2011)
14. Post, A., Menzel, I., Podelski, A.: Applying restricted english grammar on automotive requirements — does it work? a case study. In: REFSQ, pp. 166–180 (2011)
15. Skakkebæk, J.: Liveness and Fairness in Duration Calculus. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 283–298. Springer, Heidelberg (1994)

16. Walia, G.S., Carver, J.C.: A systematic literature review to identify and classify software requirement errors. Inf. Softw. Technol. 51(7), 1087–1109 (2009)
17. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering: an introduction. Kluwer Acad. Pub., Norwell (2000)
18. Yu, L., Su, S., Luo, S., Su, Y.: Completeness and consistency analysis on requirements of distributed event-driven systems. In: TASE, Washington, pp. 241–244 (2008)
19. Zhou, C., Hansen, M.: Duration Calculus: A Formal Approach to Real-Time Systems. Springer, Heidelberg (2004)

Our Experience with the CodeContracts Static Checker (Invited Tutorial)

Francesco Logozzo

Microsoft Research, Redmond, WA, USA
logozzo@microsoft.com

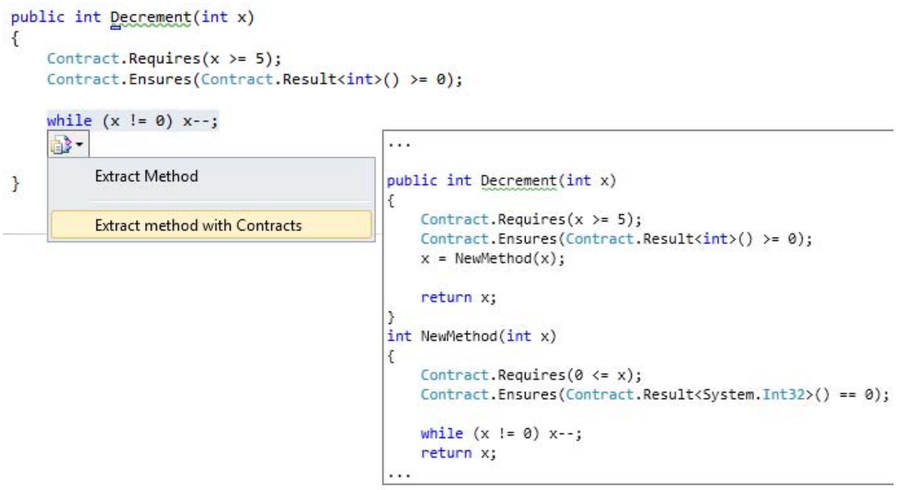
In this tutorial I will report our experience with CodeContracts [5], and in particular with its static checker (cccheck/clousot) [6].

CodeContracts are a language-agnostic solution to the specification problem. Preconditions, postconditions and object invariants are with opportune method calls acting as specification markers [4]. The CodeContracts API is part of the core .NET standard. The CodeContracts tools have been downloaded more than 50 000 times, and they are currently used in many projects by professional programmers.

The CodeContracts static checker (cccheck) is designed to be used by non-expert professional programmers, with no background in formal methods, in their every-day development activity. The evolution of cccheck is strongly influenced by the user community, who suggest improvements and new features but who also do not hesitate to criticize or stress the tool. Because of that, cccheck has a very pragmatic angle, and in many things it is surprisingly different from what one may expect from a purely Academic perspective.

The main difference of cccheck with respect to similar tools is that is based on abstract interpretation [1]. It focuses on the properties of main interest for the programmer (e.g., non-null values, numerical relationship [9,7], floating point comparisons [10], collection contents [2], and simple universally and existential quantifiers . . .). It infers loop invariants, preconditions [3], postconditions, object invariants [8]; it makes explicit the assumptions in the code as the programmer type in her program. We found inference to be a crucial point for the adoption of the tool. Whereas a strict Design-by-Contract discipline requires the programmer to provide all the annotations, in practice very few of them are willing to pay the burden of the full annotation process.

In the tutorial I detail the internals of, and the experience with cccheck. I report (some of) the user feedback (in the good and the bad). I conclude with a vision of cccheck as a real-time semantic programmer assistant, suggesting semantic code fixes, improving the refactoring experience (e.g., with the "extract method with contracts" refactoring of Fig. 1), and acting as a discrete, non-intrusive program verifier. The verbosity of the warnings can be finely tuned and only a certain class of warnings can be shown (e.g., "all callers that do not satisfy this precondition").



```
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);

    while (x != 0) x--;

}
```

```
...

public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);
    x = NewMethod(x);

    return x;
}
int NewMethod(int x)
{
    Contract.Requires(0 <= x);
    Contract.Ensures(Contract.Result<System.Int32>() == 0);

    while (x != 0) x--;
    return x;
...
```

Fig. 1. A screenshot of the extract method with contracts. The suggested contract for the extracted method is correct, safe, complete and the most general one.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977. ACM Press (1977)
2. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
3. Cousot, P., Cousot, R., Logozzo, F.: Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 150–168. Springer, Heidelberg (2011)
4. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2103–2110. ACM, New York (2010)
5. Fähndrich, M., Barnett, M., Logozzo, F.: Code Contracts (March 2009)
6. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
7. Laviron, V., Logozzo, F.: Subpolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
8. Logozzo, F.: Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 283–298. Springer, Heidelberg (2007)
9. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: SAC, pp. 184–188 (2008)
10. Logozzo, F., Fähndrich, M.: Checking compatibility of bit sizes in floating point comparison operations. In: 3rd workshop on Numerical and Symbolic Abstract Domains. ENTCS (2011)

Isabelle/*Circus*: A Process Specification and Verification Environment

Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff

Univ. Paris-Sud, Laboratoire LRI, UMR8623,
Orsay, F-91405, France
CNRS, Orsay, F-91405, France
{Abderrahmane.Feliachi,Marie-Claude.Gaudel,Burkhart.Wolff}@lri.fr

Abstract. The *Circus* specification language combines elements for complex data and behavior specifications, using an integration of Z and CSP with a refinement calculus. Its semantics is based on Hoare and He's unifying theories of programming (UTP).

We develop a machine-checked, formal semantics based on a "shallow embedding" of *Circus* in Isabelle/UTP (our semantic theory of UTP based on Isabelle/HOL). We derive proof rules from this semantics and implement tactic support that finally allows for proofs of refinement for *Circus* processes (involving both data and behavioral aspects).

This proof environment supports a syntax for the semantic definitions which is close to textbook presentations of *Circus*.

Keywords: *Circus*, denotational semantics, Isabelle/HOL, Process Algebras, Refinement.

1 Introduction

Many systems involve both complex (sometimes infinite) data structures and interactions between concurrent processes. Refinement of abstract specifications of such systems into more concrete ones, requires an appropriate formalisation of refinement and appropriate proof support.

There are several combinations of process-oriented modeling languages with data-oriented specification formalisms such as Z or B or CASL; examples are discussed in [3,10,17,14]. In this paper, we consider *Circus* [18], a language for refinement, that supports modeling of high-level specifications, designs, and concrete programs. It is representative of a class of languages that provide facilities to model data types, using a predicate-based notation, and patterns of interactions, without imposing architectural restrictions. It is this feature that makes it suitable for reasoning about both abstract and low-level designs.

We present a "shallow embedding" of the *Circus* semantics enabling state variables and channels in *Circus* to have arbitrary HOL types. Therefore, the entire handling of typing can be completely shifted to the (efficiently implemented) Isabelle type-checker and is therefore implicit in proofs. This drastically simplifies definitions and proofs, and makes the reuse of standardized proof procedures possible. Compared to implementations based on a "deep embedding" such as [19] this significantly improves the usability of the resulting proof environment.

Our representation brings particular technical challenges and contributions concerning some important notions about variables. The main challenge was to represent alphabets and bindings in a typed way that preserves the semantics and improves deduction. We provide a representation of bindings without an explicit management of alphabets. However, the representation of some core concepts in the unifying theories of programming (UTP) and *Circus* constructs (variable scopes and renaming) became challenging. Thus, we propose a (stack-based) solution that allows the coding of state variables scoping with no need for renaming. This solution is even a contribution to the UTP theory that does not allow nested variable scoping. Some challenging and tricky definitions (e.g. channels and name sets) are explained in this paper.

This paper is organized as follows. The next section gives an introduction to the basics of our work: Isabelle/HOL, UTP and *Circus* with a short example of a *Circus* process. In Section 3, we present our embedding of the basic concepts of *Circus* (alphabet, variables ...). We introduce the representation of some *Circus* actions and process, with an overview of the Isabelle/*Circus* syntax. In Section 4, we show on an example, how Isabelle/*Circus* can be used to write specifications. We give some details on what is happening "behind the scenes" when the system parses each part of the specification. In the last part of this section, we show how to write proofs based on specifications, and give a refinement proof example. A more developed version of this paper can be found in [9].

2 Background

2.1 Isabelle, HOL and Isabelle/HOL

Isabelle [12] is a generic theorem prover implemented in SML. It is based on the so-called "LCF-style architecture", which makes it possible to extend a small trusted logical kernel by user-programmed procedures in a logically safe way. New object logics can be introduced to Isabelle by specifying their syntax and semantics, by deriving its inference rules from there and program specific tactic support for the object logic. Isabelle is based on a typed λ-calculus including a Haskell-style type-system with type-classes (e.g. in α :: order, the type-variable ranges over all types that posses a partial ordering.)

Higher-Order Logic (HOL) [7,1] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \Rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x \bullet P\ x$ and $\exists x \bullet P\ x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow$ bool. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is an instance of Isabelle with higher-order logic. It provides a rich collection of library theories like sets, pairs, relations, partial functions lists, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative, i. e. logically safe definitions. Setups for the automated proof procedures like `simp`, `auto`, and arithmetic types such as `int` are provided.

2.2 Advanced Specification Constructs in Isabelle/HOL

Constant Definitions. In its easiest form, constant definitions are definitional logical axioms of the form $c \equiv E$ where c is a fresh constant symbol not occurring in E which is closed (both wrt. variables and type variables). For example:

```
definition upd::(α⇒β)⇒α⇒β⇒(α⇒β)     ("_(|_ := _|)")
where      upd f x v ≡ λ z. if x=z then v else f z
```

The pragma `("_(| _ := _|)")` for the Isabelle syntax engine introduces the notation `f(|x:=y|)` for `upd f x y`. Moreover, some elaborate preprocessing allows for recursive definitions, provided that a termination ordering can be established. Such recursive definitions are thus internally reduced to definitional axioms.

Type Definitions. Types can be introduced in Isabelle/HOL in different ways. The most general way to safely introduce new types is using the `typedef` construct. This allows introducing a type as a non-empty subset of an existing type. More precisely, the new type is specified to be isomorphic to this non-empty subset. For instance:

```
typedef mytype = "{x::nat. x < 10}"
```

This definition requires that the set is non-empty: `∃x. x∈{x::nat. x<10}`, which is easy to prove in this case:

```
by (rule_tac x = 1 in exI, simp)
```

where *`rule_tac`* is a tactic that applies an introduction rule, and `exI` corresponds to the introduction of the existential quantification.

Similarly, the `datatype` command allows the definition of inductive datatypes. It introduces a datatype using a list of *constructors*. For instance, a logical compiler is invoked for the following introduction of the type `option`:

```
datatype α option  = None | Some α
```

which generates the underlying type definition and derives distinctness rules and induction principles. Besides the *constructors* `None` and `Some`, the following match-operator and his rules are also generated:

case x of None ⇒ ... | Some a ⇒ ...

Extensible Records. Isabelle/HOL's support for *extensible records* is of particular importance for our work. Record types are denoted, for example, by:

```
record T = a::T₁
           b::T₂
```

which implicitly introduces the record constructor $(\!|$`a:=`e_1`,b:=`$e_2|\!)$ and the update of record r in field a, written as `r`$(\!|$`a:= x`$|\!)$. Extensible records are represented internally by cartesian products with an implicit free component δ, i.e. in this case by a triple of the type $T_1 \times T_2 \times \delta$. The third component can be referenced by a *special selector* `more` available on extensible records. Thus, the record `T` can be extended later on using the syntax:

```
record ET = T + c::T3
```

The key point is that theorems can be established, once and for all, on `T` types, even if future parts of the record are not yet known, and reused in the later definition and proofs over `ET`-values. Using this feature, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining the full expressivity of HOL wrt. the types of T_1, T_2 and T_3.

2.3 *Circus* and Its UTP Foundation

Circus is a formal specification language [18] which integrates the notions of states and complex data types (in a Z-like style) and communicating parallel processes inspired from CSP. From Z, the language inherits the notion of a schema used to model sets of (ground) states as well as syntactic machinery to describe pre-states and post-states; from CSP, the language inherits the concept of *communication events* and typed communication channels, the concepts of deterministic and non-deterministic choice (reflected by the process combinators $P \,\Box\, P'$ and $P \sqcap P'$), the concept of concealment (hiding) $P \backslash A$ of events in A occurring in in the evolution of process P. Due to the presence of state variables, the *Circus* synchronous communication operator syntax is slightly different frome CSP: $P \llbracket\ n \mid c \mid n'\ \rrbracket P'$ means that P and P' communicate via the channels mentioned in c; moreover, P may modify the variables mentioned in n only, and P' in n' only, n and n' are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics [15], (but coined in terms of the UTP [13]) providing a notion of execution trace `tr`, refusals `ref`, and divergences. It is expressed in terms of the UTP [11] which makes it amenable to other refinement-notions in UTP. Figure 1 presents a simple *Circus* specification, `FIG`, the fresh identifiers generator.

Predicates and Relations. The UTP is a semantic framework based on an alphabetized relational calculus. An *alphabetized predicate* is a pair (*alphabet*, *predicate*) where the free variables appearing in the predicate are all in the alphabet, e.g. $(\{x, y\}, x > y)$. As such, it is very similar to the concept of a *schema* in Z. In the base theory Isabelle/UTP of this work, we represent alphabetized predicates by sets of (extensible) records, e.g. `{A. x A > y A}`.

An *alphabetized relation* is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this case the predicate describes a relation between input and output variables, for example

$[ID]$

channel req
channel $ret, out : ID$

process $FIG \mathrel{\widehat=}$ **begin**
state $S \ == \ [\, idS : \mathbb{P}\ ID \,]$
$Init \mathrel{\widehat=} idS := \emptyset$

$$\begin{array}{|l}
\multicolumn{1}{l}{Out} \\
\Delta S \\
v! : ID \\
\hline
v! \notin idS \\
idS' = idS \cup \{v!\} \\
\hline
\end{array}
\qquad\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Remove} \\
\Delta S \\
x? : ID \\
\hline
idS' = idS \setminus \{x?\} \\
\hline
\end{array}$$

$\bullet \ Init \, ; \ \textbf{var}\ v : ID \bullet$
$(\mu \ X \bullet \ (req \to Out \, ; \ out!v \to Skip \ \Box \ ret?x \to Remove) \, ; \ X)$
end

Fig. 1. The Fresh Identifiers Generator in (Textbook) *Circus*

$(\{x, x', y, y'\}, x' = x + y)$ which is a notation for: `{(A,A').x A' = x A + y A}`, which is a set of pairs, thus a relation.

Standard predicate calculus operators are used to combine alphabetized predicates. The definition of these operators is very similar to the standard one, with some additional constraints on the alphabets.

Designs and Processes. In UTP, in order to explicitly record the termination of a program, a subset of alphabetized relations is introduced. These relations are called *designs* and their alphabet should contain the special boolean observational variable `ok`. It is used to record the start and termination of a program. A UTP design is defined as follows in Isabelle:

```
(P ⊢ Q) ≡ λ (A,A'). (ok A ∧ P (A,A')) ⟶ (ok A' ∧ Q (A,A'))
```

Following the way of UTP to describe reactive processes, more observational variables are needed to record the interaction with the environment. Three observational variables are defined for this subset of relations: `wait`, `tr` and `ref`. The boolean variable `wait` records if the process is waiting for an interaction or has terminated. `tr` records the list (trace) of interactions the process has performed so far. The variable `ref` contains the set of interactions (events) the process may refuse to perform. These observational variables defines the basic alphabet of all reactive processes called "`alpha_rp`".

Some healthiness conditions are defined over `wait`, `tr` and `ref` to ensure that a recative process satisfies some properties [6] (see Table 2 in [9]).

A CSP process is a UTP reactive process that satisfies two additional healthiness conditions(all well-formedness conditions can be found in [9]). A process that satisfies these conditions is said to be CSP healthy.

3 Isabelle/*Circus*

The Isabelle/*Circus* environment allows a syntax of processes which is close to the textbook presentations of *Circus* (see Fig. 2). Similar to other specification constructs in Isabelle/HOL, this syntax is "parsed away", i. e. compiled into an internal representation of the denotational semantics of *Circus*, which is a formalization in form of a shallow embedding of the (essentially untyped) paper-and-pencil definitions by Oliveira et al. [13], based on UTP. *Circus* actions are defined as CSP healthy reactive processes.

```
Process    ::= circusprocess Tpar* name = PParagraph* where Action
PParagraph ::= AlphabetP | StateP | ChannelP | NamesetP | ChansetP | SchemaP
             | ActionP
AlphabetP  ::= alphabet [ vardecl+ ]
vardecl    ::= name :: type
StateP     ::= state [ vardecl+ ]
ChannelP   ::= channel [ chandecl+ ]
chandecl   ::= name | name type
NamesetP   ::= nameset name = [ name+ ]
ChansetP   ::= chanset name = [ name+ ]
SchemaP    ::= schema name = SchemaExpression
ActionP    ::= action name = Action
Action     ::= Skip | Stop | Action ; Action | Action □ Action | Action ⊓ Action
             | Action \ chansetN | var := expr | guard & Action | comm → Action
             | Schema name | ActionName | μ var • Action | var var • Action
             | Action ⟦ namesetN | chansetN | namesetN ⟧ Action
```

Fig. 2. Isabelle/*Circus* syntax

In the UTP representation of reactive processes we have given in a previous paper [8], the process type is generic. It contains two type parameters that represent the channel type and the alphabet of the process. These parameters are very general, and they are instantiated for each specific process. This could be problematic when representing the *Circus* semantics, since some definitions rely directly on variables and channels (e.g assignment and communication). In this section we present our solution to deal with this kind of problems, and our representation of the *Circus* actions and processes.

We now describe the foundation as well as the semantic definition of some process operators of *Circus*. A distinguishing feature of *Circus* processes are explicit state variables which do not exist in other process algebras like, e.g., CSP. These can be:

- *global* state variables, i. e. they are declared via alphabetized predicates in the `state` section, or Z-like Δ operations on global states that generate alphabetized relations, or
- *local* state variables, i. e. they are result of the variable declaration statement **var** var • Action. The scope of local variables is restricted to Action.

On both kind of state variables, logical constraints may be expressed.

3.1 Alphabets and Variables

In order to define the set of variables of a specification, the *Circus* semantics considers the alphabet of its components, be it on the level of alphabetized predicates, alphabetized relations or actions. We recall that these items are represented by sets of records or sets of pairs of records. The *alphabet of a process* is defined by extending the basic reactive process alphabet (cf. Section 2.3) by its variable names and types. For the example *FIG*, where the global state variable *idS* is defined, this is reflected in Isabelle/Circus by the extension of the process alphabet by this variable, i.e. by the extension of the Isabelle/HOL record:

```
record α alpha = α alpha_rp +  idS :: ID set
```

This introduces the record type `alpha` that contains the observational variables of a reactive process, plus the variable `idS`. Note that our *Circus* semantic representation allows "built-in" bindings of alphabets in a typed way. Moreover, there is no restriction on the associated HOL type. However, the inconvenience of this representation is that variables cannot be introduced "on the fly"; they must be known statically i.e. at type inference time. Another consequence is that a "syntactic" operation such as variable renaming has to be expressed as a "semantic" operation that maps one record type into another.

Updating and Accessing Global Variables. Since the alphabets are represented by HOL records, i.e. a kind binding "*name* $\mapsto$ *value*", we need a certain infrastructure to access data in them and to update them. The Isabelle representation as records gives us already two functions (for each record) "select" and "update". The "select" function returns the value of a given variable name, and the "update" functions updates the value of this variable. Since we may have different HOL types for different variables, a unique definition for select and update cannot be provided. There is an instance of these functions for each variable in the record. The name of the variable is used to distinguish the different instances: for the select function the name is used directly and for the update function the name is used as a prefix e.g. for a variable named "x" the names of the *select* and *update* functions are respectively `x` of type α and `x_update`.

Since a variable is characterized essentially by these functions, we define a general type (synonym) called `var` which represents a variable as a pair of its select and update function (in the underlying state σ).

```
 types (β, σ) var = "(σ ⇒ β) * ((β ⇒ β) ⇒ σ ⇒ σ)"
```

For a given alphabet (record) of type σ, `(β, the type σ)var` represents the type of the variables whose value type is β. One can then extract the select and update functions from a given variable with the following functions:

```
 definition select :: "(β, σ) var ⇒σ ⇒ β"
   where select f ≡ (fst f)

 definition update :: "(β, σ) var ⇒ β ⇒ σ ⇒ σ "
   where update f v ≡ (snd f) (λ _ . v)
```

Finally, we introduce a function called VAR to implement a syntactic translation of a variable name to an entity of type var.

```
syntax "_VAR" :: "id ⇒(β, σ) var" ("VAR _")
translations VAR x => (x, _update_ name x)
```

Note that in this syntactic translation rule, _update_ name x stands for the concatenation of the string _update_ with the content of the variable x; the resulting _update_x in this example is mapped to the field-update function of the extensible record x_update by a default mechanism. On this basis, the assignment notation can be written as usual:

```
syntax
  "_assign" :: "id ⇒(σ ⇒β) ⇒(α, σ) action" ("_ ‘:=‘ _")
translations
  "x ‘:=‘ E"  => "CONST ASSIGN (VAR x) E"
```

and mapped to the *semantics* of the program variable (x,x_update) together with the universal ASSIGN operator defined later on, in Section 3.3.

Updating and Accessing Local Variables. In *Circus*, local program variables can be introduced on the fly, and their scopes are explicitly defined, as can be seen in the *FIG* example. In textbook *Circus*, nested scopes are handled by variable renaming which is not possible in our representation due to the implicit representation of variable names. We represent local program variables by global variables, using the var type defined above, where selection and update involve an explicit stack discipline. Each variable is mapped to a list of values, and not to one value only (as for state variables). Entering the scope of a variable is just adding a new value as the head of the corresponding values list. Leaving a variable scope is just removing the head of the values list. The select and update functions correspond to selecting and updating the head of the list. This ensures dynamic scoping, as it is stated by the *Circus* semantics.

Note that this encoding scheme requires to make local variables lexically distinct from global variables; local variable instances are just distinguished from the global ones by the stack discipline.

3.2 Synchronization Infrastructure: Name Sets and Channels

Name Sets. An important notion, used in the definition of parallel *Circus* actions, is name sets as seen in Section 2.3. A name set is a set of variable names, which is a subset of the alphabet. This notion cannot be directly expressed in our representation since variable names are not explicitly represented. Thus its definition relies on the characterization of the variables in our representation. As for variables, name sets are defined by their functional characterization. They are used in the definition of the binding merge function *MSt* below:
$\forall v \bullet (v \in ns1 \Rightarrow v' = (1.v)) \wedge (v \in ns2 \Rightarrow v' = (2.v)) \wedge (v \notin ns1 \cup ns2 \Rightarrow v' = v).$

The disjoint name sets $ns1$ and $ns2$ are used to determine which variable values (extracted from local bindings of the parallel components) are used to update the global binding of the process. A name set can be functionally defined as a binding update function, that copies values from a local binding to the global one. For example, a name set NS that only contains the variable x can be defined as follows in Isabelle/Circus:

```
definition NS lb gb ≡ x_update (x lb) gb
```

where `lb` and `gb` stands for local and global bindings, `x` and `x_update` are the select and update functions of variable `x`. Then the merge function can be defined by composing the application of the name sets to the global binding.

Channels. Reactive processes interact with the environment via synchronizations and communications. A synchronization is an interaction via a channel without any exchange of data. A communication is a synchronization with data exchange. In order to reason about communications in the same way, a datatype *channels* is defined using the channels names as constructors. For instance, in:

```
datatype channels = chan1 | chan2 nat | chan3 bool
```

we declare three channels: `chan1` that synchronizes without data , `chan2` that communicates natural values and `chan3` that exchanges boolean values.

This definition makes it possible to reason globally about communications since they have the same type. However, the channels may not have the same type: in the example above, the types of `chan1`, `chan2` and `chan3` are respectively `channels`, `nat ⇒channels` and `bool ⇒channels`. In the definition of some *Circus* operators, we need to compare two channels, and one can't compare for example `chan1` with `chan2` since they don't have the same type. A solution would be to compare `chan1` with (`chan2 v`). The types are equivalent in this case, but the problem remains because comparing (`chan2 0`) to (`chan2 1`) will state inequality just because the communicated values are not equal. We could define an inductive function over the datatype `channels` to compare channels, but this is only possible when all the channels are known *a priori*.

Thus, we add some constraint to the generic channels type: we require the `channels` type to implement a function `chan_eq` that tests the equality of two channels. Fortunately, Isabelle/HOL provides a construct for this kind of restriction: the type classes (sorts) mentioned in Section 2.1. We define a type class (interface) `chan_eq` that contains a signature of the `chan_eq` function.

```
class chan_eq =
 fixes chan_eq :: "α ⇒α ⇒ bool"
begin end
```

Concrete channels type must implement the interface (class) " `chan_eq`" that can be easily defined for this concrete type. Moreover, one can use this class to add some definition that depends on the channel equivalence function. For example, a trace equivalence function can be defined as follows:

```
fun tr_eq where
  tr_eq [] [] = True | tr_eq xs [] = False | tr_eq [] ys = False
| tr_eq (x#xs) (y#ys) = if chan_eq x y then tr_eq xs ys else False
```

It is applicable to traces of elements whose type belongs to the sort `chan_eq`.

3.3 Actions and Processes

The *Circus* actions type is defined as the set of all the CSP healthy reactive processes. The type `(α,σ)relation_rp` is the reactive process type where α is of `channels` type and σ is a record extensions of `action_rp`, i. e. the global state variables. On this basis, we can encode the concept of a process for a family of possible state instances. We introduce below the vital type `action`:

```
typedef(Action)
 (α::chan_eq,σ) action = {p::(α,σ)relation_rp. is_CSP_process p}
proof - {...}
qed
```

As mentioned before, a type-definition introduces a new type by stating a set. In our case it is the set of reactive processes that satisfy the healthiness-conditions for CSP-processes, isomorphic to the new type.

Technically, this construct introduces two constants definitions `Abs_Action` and `Rep_Action` respectively of type `(α,σ) relation_rp ⇒(α,σ) action` and `(α,σ)action ⇒(α,σ)relation_rp` as well as the usual two axioms expressing the bijection `Abs_Action(Rep_Action(X))=X` and `is_CSP_process p ⟹ Rep_Action(Abs_Action(p))=p` where `is_CSP_process` captures the healthiness conditions.

Every *Circus* action is an abstraction of an alphabetized predicate. In [9], we introduce the definitions of all the actions and operators using their denotational semantics. The environment contains, for each action, the proof that this predicate is CSP healthy. In this section, we present some of the important definitions, namely: basic actions, assignments, communications, hiding, and recursion.

Basic Actions. `Stop` is defined as a reactive design, with a precondition `true` and a postcondition stating that the system deadlocks and the traces are not evolving.

```
definition
Stop ≡  Abs_Action (R (true ⊢λ(A, A'). tr A' = tr A ∧wait A'))
```

`Skip` is defined as a reactive design, with a precondition *true* and a postcondition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the `more` field (seen in Section 2.2) is not changed, since this field is mapped to all the state variables. Note that using the `more`-field is a tribute to our encoding of alphabets by extensible records and stands for all future extensions of the alphabet (e.g. state variables).

```
definition Skip ≡  Abs_Action (R (true ⊢ λ (A, A'). tr A' = tr A
                                          ∧ ¬ wait A' ∧ more A = more A'))
```

The Universal Assignment Action. In Section 3.1, we described how global and local variables are represented by access- and updates functions introduced by fields in extensible records. In these terms, the "lifting" to the assignment action in *Circus* processes is straightforward:

```
definition
  ASSIGN::"(β, σ) var ⇒(σ ⇒ β) ⇒(α::ev_eq, σ) action"
where
  ASSIGN x e ≡ Abs_Action (R (true ⊢ Y))
where
 Y = λ(A, A'). tr A' = tr A ∧ ¬wait A' ∧
                more A' = (assign x (e (more A))) (more A)
```

where `assign` is the projection into the update operation of a semantic variable described in section 3.1.

Communications. The definition of prefixed actions is based on the definition of a special relation `do_I`. In the *Circus* denotational semantics [13], various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

```
definition do_I c x P ≡  X  ◁ wait o fst ▷  Y
where
X = (λ (A, A'). tr A = tr A' ∧ ((c ‘ P) ∩ ref A') = {})
and
Y = (λ (A, A'). hd ((tr A') - (tr A)) ∈ (c ‘ P) ∧
     (c (select x (more A))) = (last (tr A')))
```

where `c` is a channel constructor, `x` is a variable (of `var` type) and `P` is a predicate. The `do_I` relation gives the semantics of an interaction: if the system is ready to interact, the trace is unchanged and the waiting channel is not refused. After performing the interaction, the new event in the trace corresponds to this interaction.

The semantics of the whole action is given by the following definition:

```
definition Prefix c x P S ≡Abs_Action(R (true ⊢ Y)) ; S
where
Y =  do_I c x P ∧  (λ (A, A'). more A' = more A)
```

where `c` is a channel constructor, `x` is a variable (of type var), `P` is a predicate and `S` is an action. This definition states that the prefixed action semantics is given by the interaction semantics (`do_I`) sequentially composed with the semantics of the continuation (action `S`).

Different types of communication are considered:

– Inputs: the communication is done over a variable.
– Constrained Inputs: the input variable value is constrained with a predicate.
– Outputs: the communications exchanges only one value.
– Synchronizations: only the channel name is considered (no data).

The semantics of these different forms of communications is based on the general definition above.

```
definition read c x P ≡ Prefix c x true P
definition write1 c a P ≡ Prefix c (λs. a s, (λ x. λy. y)) true P
definition write0 c P ≡ Prefix (λ_.c) (λ_._, (λ x. λy. y)) true P
```

where `read`, `write1` and `write0` respectively correspond to inputs, outputs and synchronization. Constrained inputs correspond to the general definition.

We configure the Isabelle syntax-engine such that it parses the usual communication primitives and gives the corresponding semantics:

```
translations
  c ? p →P      == CONST read c (VAR p) P
  c ? p : b →P  == CONST Prefix c (VAR p) b P
  c ! p →P      == CONST write1 c p P
  a → P         == CONST write0 (TYPE(_)) a P
```

Hiding. The hiding operator is interesting because it depends on a channel set. This operator `P \ cs` is used to encapsulate the events that are in the channel set `cs`. These events become no longer visible from the environment. The semantics of the hiding operator is given by the following reactive process:

```
definition
Hide ::"[(α, σ) action , α set] ⇒(α, σ) action" (infixl "\")
where
P \ cs ≡Abs_Action( R(λ (A, A').
          ∃ s. (Rep_Action P)(A, A'⦇tr :=s, ref := (ref A') ∪ cs⦈)
               ∧ (tr A' - tr A) = (tr_filter (s - tr A) cs))); Skip
```

The definition uses a filtering function `tr_filter` that removes from a trace the events whose channels belong to a given set. The definition of this function is based on the function `chan_eq` we defined in the class `chan_eq`. This explains the presence of the constraint on the type of the action channels in the hiding definition, and in the definition of the filtering function below:

```
fun tr_filter::"a::chan_eq list ⇒a set ⇒a list" where
  tr_filter [] cs = []
| tr_filter (x#xs) cs = (if (¬ chan-in_set x cs)
                           then (x#(tr_filter xs cs))
                             else (tr_filter xs cs))
```

where the `chan-in_set` function checks if a given channel belongs to a channel set using `chan_eq` as equality function.

Recursion. To represent the recursion operator "μ" over actions, we use the universal least fix-point operator "*lfp*" defined in the HOL library for lattices and we follow again [13]. The use of least fix-points in [13] is the most substantial deviation from the standard CSP denotational semantics, which requires Scott-domains and complete partial orderings. The operator *lfp* is inherited from the

"*Complete Lattice class*" under some conditions, and all theorems defined over this operator can be reused. In order to reuse this operator, we have to show that the least-fixpoint over functionals that enrich pairs of failure - and divergence trace sets monotonely, produces an `action` that satisfies the CSP healthiness conditions. This consistency proof for the recursion operator is the largest contained in the Isabelle/*Circus* library.

Therefore, we must prove that the *Circus* actions type defines a complete lattice. This leads to prove that the actions type belongs to the HOL "*Complete Lattice class*". Since type classes in HOL are hierarchic, the proof is in three steps: first, a proof that the *Circus* actions type forms a lattice by instantiating the HOL "*Lattice class*"; second, a proof that actions type instantiates a subclass of lattices called "*Bounded Lattice class*"; third, proof of the instantiation from the "*Complete Lattice class*". More on these proofs can be found in [9].

***Circus* Processes.** A *Circus* process is defined in our environment as a local theory by introducing qualified names for all its components. This is very similar to the notion of *namespaces* popular in programming languages. Defining a *Circus* process locally makes it possible to encapsulate definitions of alphabet, channels, schema expressions and actions in the same namespace. It is important for the foundation of Isabelle/*Circus* to avoid the ambiguity between local process entities definitions (e.g. `FIG.Out` and `DFIG.Out` in the example of Section 4).

4 Using Isabelle/*Circus*

We describe the front-end interface of Isabelle/*Circus*. In order to support a maximum of common *Circus* syntactic look-and-feel, we have programmed at the SML level of Isabelle a compiler that parses and (partially) pretty prints *Circus* process given in the syntax presented in Figure 2.

4.1 Writing Specifications

A specification is a sequence of paragraphs. Each paragraph may be a declaration of alphabet, state, channels, name sets, channel sets, schema expressions or actions. The main action is introduced by the keyword `where`. Below, we illustrate how to use the environment to write a *Circus* specification using the `FIG` process example presented in Figure 1.

```
circusprocess FIG =
  alphabet = [v::nat, x::nat]
  state = [idS::nat set]
  channel = [req, ret nat, out nat]
  schema Init = idS := {}
  schema Out = ∃a. v’ = a ∧ v’ ∉ idS ∧ idS’ = idS ∪{v’}
  schema Remove = x ∉ idS ∧ idS’ = idS - {x}
  where var v • Schema Init; (μ X •(req →Schema Out; out!v →Skip)
                                  □ (ret?x →Schema Remove); X)
```

Each line of the specification is translated into the corresponding semantic operator given in Section 3.3. We describe below the result of executing each command of `FIG`:

– the compiler introduces a scope of local components whose names are qualified by the process name (`FIG` in the example).
– `alphabet` generates a list of record fields to represent the binding. These fields map names to value lists.
– `state` generates a list of record fields that corresponds to the state variables. The names are mapped to single values. This command, together with `alphabet` command, generates a record that represents all the variables (for the `FIG` example the command generates the record `FIG_alphabet`, that contains the fields `v` and `x` of type `nat list` and the field `idS` of type `nat set`).
– `channel` introduces a datatype of typed communication channels (for the `FIG` example the command generates the datatype `FIG_channels` that contains the constructors `req` without communicated value and `ret` and `out` that communicate natural values).
– `schema` allows the definition of schema expressions represented as an alphabetized relation over the process variables (in the example the schema expressions `FIG.Init`, `FIG.Out` and `FIG.Remove` are generated).
– `action` introduces definitions for *Circus* actions in the process. These definitions are based on the denotational semantics of *Circus* actions. The type parameters of the action type are instantiated with the locally defined channels and alphabet types.
– `where` introduces the main action as in `action` command (in the example the main action is `FIG.FIG` of type `(FIG_channels, FIG_alphabet)action`).

4.2 Relational and Functional Refinement in Circus

The main goal of Isabelle/*Circus* is to provide a proof environment for *Circus* processes. The "shallow-embedding" of *Circus* and UTP in Isabelle/HOL offers the possibility to reuse proof procedures, infrastructure and theorem libraries already existing in Isabelle/HOL. Moreover, once a process specification is encoded and parsed in Isabelle/*Circus*, proofs of, e. g., refinement properties can be developped using the ISAR language for structured proofs.

To show in more details how to use Isabelle/*Circus*, we provide a small example of action refinement proof. The refinement relation is defined as the universal reverse implication in the UTP. In *Circus*, it is defined as follows:

```
definition A1 ⊑c A2 ≡(Rep_Action A1) ⊑utp (Rep_Action A2)
```

where A1 and A2 are *Circus* actions, `⊑c` and `⊑utp` stands respectively for refinement relation on *Circus* actions and on UTP predicate.

This definition assumes that the actions A1 and A2 share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [16,5] by backwards and forwards simulations. In this paper, we restrict ourselves to a special case, the so-called *functional* backwards simulation. This refers to the fact that the abstraction relation R that relates concrete and abstract actions is just a function:

```
definition Simulation ("_ ≼_ _") where
 A1 ≼R A2 = ∀a b.(Rep_Action A2)(a,b) ⟶(Rep_Action A1)(R a,R b)
```

where A1 and A2 are *Circus* actions and R is a function mapping the corresponding A1 alphabet to the A2 alphabet.

4.3 Refinement Proofs

We can use the definition of simulation to transform the proof of refinement to a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. The problem with this approach is that the size of proofs will grow exponentially with the size of the processes. To avoid this problem, some general refinement laws were defined in [5] to deal with the refinement of *Circus* actions at operators level and not at UTP level. We introduced and proved a subset of theses laws in our environment (see Table 1).

Table 1. Proved refinement laws

$\dfrac{P \preceq_S Q \qquad P' \preceq_S Q'}{P;\ P' \preceq_S Q;\ Q'}$ SeqI	$\dfrac{P \preceq_S Q \qquad g_1 \simeq_S g_2}{g_1 \& P \preceq_S g_2 \& Q}$ GrdI
$\dfrac{P \preceq_S Q \qquad x \sim_S y}{var\ x \bullet P \preceq_S var\ y \bullet Q}$ VarI	$\dfrac{P \preceq_S Q \qquad x \sim_S y}{c?x \to P \preceq_S c?y \to Q}$ InpI
$\dfrac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \sqcap P' \preceq_S Q \sqcap Q'}$ NdetI	$\dfrac{P \preceq_S Q \qquad x \sim_S y}{c!x \to P \preceq_S c!y \to Q}$ OutI
$\dfrac{\begin{array}{c}[X \preceq_S Y] \\ \vdots \\ P\ X \preceq_S Q\ Y\end{array} \qquad mono\ P \qquad mono\ Q}{\mu\ X \bullet P\ X \preceq_S \mu\ Y \bullet Q\ Y}$ MuI	$\dfrac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \Box P' \preceq_S Q \Box Q'}$ DetI
$\dfrac{\begin{array}{c}[Pre\ sc_1\ (S\ A)] \\ \vdots \\ Pre\ sc_2\ A\end{array} \qquad \begin{array}{c}[Pre\ sc_1\ (S\ A) \quad sc_2\ (A, A')] \\ \vdots \\ sc_1\ (S\ A, S\ A')\end{array}}{schema\ sc_1 \preceq_S schema\ sc_2}$ SchI	$\dfrac{P \preceq_S Q}{a \to P \preceq_S a \to Q}$ SyncI
$\dfrac{P \preceq_S Q \qquad P' \preceq_S Q' \qquad ns_1 \sim_S ns_1' \qquad ns_2 \sim_S ns_2'}{P [\![ns_1 \mid cs \mid ns_2]\!] P' \preceq_S Q [\![ns_1' \mid cs \mid ns_2']\!] Q'}$ ParI	$\dfrac{}{Skip \preceq_S Skip}$ SkipI

In Table 1, the relations "$x \sim_S y$" and "$g_1 \simeq_S g_2$" record the fact that the variable x (repectively the guard g_1) is refined by the variable y (repectively by the guard g_2) w.r.t the simulation function S.

These laws can be used in complex refinement proofs to simplify them at the *Circus* level. More rules can be defined and proved to deal with more complicated statements like combination of operators for example. Using these laws, and exploiting the advantages of a shallow embedding, the automated proof of refinement becomes surprisingly simple.

Coming back to our example, let us consider the DFIG specification below, where the management of the identifiers via the set idS is refined into a set of removed identifiers retidS and a number max, which is the rank of the last issued identifier.

```
circusprocess DFIG =
  alphabet = [w::nat, y::nat]
  state = [retidS::nat set, max::nat]
  schema Init = retidS' = {} ∧max' = 0
  schema Out = w' = max ∧ max' = max+1 ∧ retidS' = retidS - {max}
  schema Remove = y < max ∧ y ∉ retidS ∧  retidS' = retidS ∪ {y}
                    ∧ max' = max
  where var w · Schema Init; (μ X ·(req →Schema Out; out!w →Skip)
                                 □ (ret?y →Schema Remove); X)
```

We provide the proof of refinement of FIG by DFIG just instantiating the simulation function R by the following abstraction function, that maps the underlying concrete states to abstract states:

```
definition Sim A = FIG_alphabet.make (w A) (y A)
                                 ({a. a < (max A) ∧ a ∉(retidS A)})
```

where A is the alphabet of DFIG, and FIG_alphabet.make yields an alphabet of type FIG_Alphabet initializing the values of v, x and idS by their corresponding values from DFIG_alphabet: w, y and {a. a < max ∧ a ∉retidS}).

To prove that DFIG is a refinement of FIG one must prove that the main action DFIG.DFIG refines the main action FIG.FIG. The definition is then simplified, and the refinement laws are applied to simplify the proof goal. Thus, the full proof consists of a few lines in ISAR:

```
theorem "FIG.FIG ≼Sim DFIG.DFIG"
  apply (auto simp: DFIG.DFIG_def FIG.FIG_def mono_Seq
              intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI)
  apply (simp_all add: SimRemove SimOut SimInit Sim_def)
done
```

First, the definitions of FIG.FIG and DFIG.DFIG are simplified and the defined refinement laws are used by the auto tactic as introduction rules. The second step replaces the definition of the simulation function and uses some proved lemmas to finish the proof. The three lemmas used in this proof: SimInit, SimOut and SimRemove give proofs of simulation for the schema Init, Out and Remove.

5 Conclusions

We have shown for the language *Circus*, which combines data-oriented modeling in the style of Z and behavioral modeling in the style of CSP, a semantics in form of a shallow embedding in Isabelle/HOL. In particular, by representing the somewhat non-standard concept of the *alphabet* in UTP in form of extensible records in HOL, we achieved a fairly compact, typed presentation of the language. In contrast to previous work based on some deep embedding [19], this shallow embedding allows arbitrary (higher-order) HOL-types for channels, events, and state-variables, such as, e.g., sets of relations etc. Besides, systematic renaming of local variables is avoided by compiling them essentially to global variables using a stack of variable instances. The necessary proofs for showing that the definitions are consistent — i. e. satisfy altogether `is_CSP_healthy` — have been done, together with a number of algebraic simplification laws on *Circus* processes.

Since the encoding effort can be hidden behind the scene by flexible extension mechanisms of the Isabelle, it is possible to have a compact notation for both specifications and proofs. Moreover, existing standard tactics of Isabelle such as `auto`, `simp` and `metis` can be reused since our *Circus* semantics is representationally close to HOL. Thus, we provide an environment that can cope with combined refinements concerning data and behavior. Finally, we demonstrate its power — w.r.t. both expressivity and proof automation — with a small, but prototypic example of a process-refinement.

In the future, we intend to use Isabelle/*Circus* for the generation of test-cases, on the basis of [4], using the HOL-TestGen-environment [2].

Acknowledgement. We warmly thank Markarius Wenzel for his valuable help with the Isabelle framework. Furthermore, we are greatly indebted to Ana Cavalcanti for her comments on the semantic foundation of this work.

References

1. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. Kluwer Academic (2002); now published by Springer
2. Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Aspects of Computing (to appear, 2012)
3. Butler, M.: CSP2B: A practical approach to combining CSP and B. Formal Aspects of Computing 12, 182–196 (2000)
4. Cavalcanti, A., Gaudel, M.-C.: Testing for refinement in Circus. Acta Informatica 48(2), 97–147 (2011)
5. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2-3), 146–181 (2003)
6. Cavalcanti, A., Woodcock, J.: A Tutorial Introduction to CSP in Unifying Theories of Programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
7. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5(2), 56–68 (1940)
8. Feliachi, A., Gaudel, M.-C., Wolff, B.: Unifying Theories in Isabelle/HOL. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 188–206. Springer, Heidelberg (2010)
9. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/Circus : a process specification and verification environment. Technical Report 1547, Univ. Paris-Sud XI LRI (November 2011), `http://www.lri.fr/srubrique.php?news=33`

10. Fischer, C.: How to Combine Z with Process Algebra. In: Bowen, J., Fett, A., Hinchey, M.G. (eds.) ZUM 1998. LNCS, vol. 1493, pp. 5–25. Springer, Heidelberg (1998)
11. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
12. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
13. Oliveira, M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A denotational semantics for Circus. Electron. Notes Theor. Comput. Sci. 187, 107–123 (2007)
14. Roggenbach, M.: CSP-CASL: a new integration of process algebra and algebraic specification. Theor. Comput. Sci. 354, 42–71 (2006)
15. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
16. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in Circus. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 451–470. Springer, Heidelberg (2002)
17. Taguchi, K., Araki, K.: The state-based CCS semantics for concurrent Z specification. In: ICFEM 1997, pp. 283–292. IEEE (1997)
18. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J., Henson, M., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
19. Zeyda, F., Cavalcanti, A.: Encoding Circus Programs in ProofPowerZ. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 218–237. Springer, Heidelberg (2010)

Termination Analysis of Imperative Programs Using Bitvector Arithmetic*

Stephan Falke[1], Deepak Kapur[2], and Carsten Sinz[1]

[1] Institute for Theoretical Computer Science, KIT, Germany
{stephan.falke,carsten.sinz}@kit.edu
[2] Dept. of Computer Science, University of New Mexico, USA
kapur@cs.unm.edu

Abstract. Currently, nearly all methods for proving termination of imperative programs apply an unsound and incomplete abstraction by treating bitvectors and bitvector arithmetic as (unbounded) integers and integer arithmetic, respectively. This abstraction ignores the wrap-around behavior caused by under- and overflows in bitvector arithmetic operations. This is particularly problematic in the termination analysis of low-level system code. This paper proposes a novel method for encoding the wrap-around behavior of bitvector arithmetic within integer arithmetic. Afterwards, existing methods for reasoning about the termination of integer arithmetic programs can be employed for reasoning about the termination of bitvector arithmetic programs. An empirical evaluation shows the practicality and effectiveness of the proposed method.

1 Introduction

Most methods for proving termination of imperative programs (e.g., [1,3,4,5,6,7,9,10,14,18,19,20,23,24] and including our own recent work [13]) developed during the past decade deviate in their analysis in one important aspect from the execution of a program on a computer: machine arithmetic operating on bitvectors of a limited range is treated as arithmetic on (unbounded) integers (or as arithmetic on real numbers). Thus, the wrap-around behavior caused by under- and overflow is ignored and the semantics of the computer program is only approximated.

This approximation is undesirable and can err in both directions:

– A program may be terminating using bitvector arithmetic, but nonterminating using integer arithmetic.
– A program may be terminating using integer arithmetic, but nonterminating using bitvector arithmetic.

Example 1. Consider the following C functions:

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

```
void f(int i)
{
    while (i > 0) {
        ++i;
    }
}
```

```
void g(int i, int j)
{
    while (i <= j) {
        ++i;
    }
}
```

Then `f` is terminating using bitvector arithmetic since `i` will eventually overflow and become negative.[1] On the other hand, `f` is nonterminating using integer arithmetic since the loop-condition stays always true whenever the argument passed to `f` is at least 1. For `g` the situation is reversed. It is nonterminating using bitvector arithmetic if `j` is `INT_MAX`, but terminating using integer arithmetic since `i` will eventually exceed `j`. ♢

This paper presents an adaptation of the method developed in [13] that correctly models the wrap-around behavior of bitvector arithmetic. For this, bitvectors are represented by integers and the wrap-around behavior is explicitly modeled in the integer domain using a normalization step after each arithmetic operation. The technique is not specific to [13], however, and can be combined with other methods for the termination analysis of imperative programs.

The only previous work concerned with the termination analysis of programs using bitvector arithmetic is, to the best of our knowledge, [8]. That paper has developed two methods for the synthesis of ranking functions for programs using bitvector arithmetic:

1. An encoding of bitvector arithmetic within integer arithmetic. This is similar in spirit to our approach, but [8] requires the use of quantifiers. These quantifiers need to be eliminated before the ranking functions can be synthesized. Quantifier elimination is an expensive operation in general. In contrast to this, the approach developed in the present paper does not introduce any quantifiers and thus does not rely on quantifier elimination.
2. An approach based on template matching for linear ranking functions. This approach uses a SAT, QBF, or (in the later [26]) SMT solver in order to instantiate the templates by solving quantified bitvector formulas of the shape $\exists x_1, \ldots, x_n.\ \forall y_1, \ldots, y_m.\ \psi$. While QBF solvers can directly be applied to such formulas, they currently lack in performance and are not very successful in solving the generated formulas. The SMT solver from [26] performs better than current QBF solvers but its performance is still not completely satisfactory. In order to apply SAT solvers to quantified bitvector formulas of the shape $\exists x_1, \ldots, x_n.\ \forall y_1, \ldots, y_m.\ \psi$, [8] uses the following approach: the

[1] Strictly speaking, a signed under- or overflow yields undefined behavior according to the C99 standard. Virtually all compilers treat signed under- and overflows using the wrap-around behavior, though. This is also the required behavior for unsigned under- and overflows according to the C99 standard. This paper assumes the wrap-around behavior semantics for both signed and unsigned under- and overflows.

existentially quantified variables $x_1, \ldots, x_n$ are instantiated by the bitvectors corresponding to values from $\{-1, 0, 1\}$, and for each of these possibilities, the instantiated quantifier-free formula $\neg\psi$ is checked for unsatisfiability. Using this approach, the SAT-based implementation of [8] is quite efficient but fails to generate ranking functions such as $2x - y$ which can easily be generated by our implementation.

An empirical comparison of our method as implemented in the tool KITTeL [13] with the methods from [8] (and [26]) is very encouraging. Of the 61 examples considered in [8] (51 terminating ones and 10 nonterminating ones), our implementation succeeds in proving termination of 38 examples. The methods from [8] (and [26]) succeed on 30 examples (quantifier-elimination-based approach), 34 examples (template-based approach with a SAT solver), 8 examples (template-based approach with a QBF solver), and 27 examples (template-based approach with the SMT solver from [26]), respectively.

The problem of unsoundness if bitvectors are abstracted to integers is also present in other static program analysis methods. Similar to termination analysis, this problem is rarely addressed, but invariant generation methods based on modular arithmetic are presented in [17,21]. Furthermore, decision procedures for modular arithmetic have been developed in [2,25].

This paper is organized as follows: Sect. 2 briefly introduces LLVM [15] since our method makes use of this compiler framework. Then, Sect. 3 recalls the method for termination analysis developed in [13], but presents it in a new light. Next, Sect. 4 shows how the wrap-around behavior of bitvector arithmetic can be modeled by introducing explicit normalization steps. Section 5 presents an empirical comparison with the methods of [8] (and [26]), and Sect. 6 concludes.

2 A Brief Overview of LLVM and Its Use in KITTeL

Termination analysis of programs written in real-life programming languages is a very important, yet notoriously difficult, task. This is in part due to the rich syntax of these languages. Furthermore, their complex and sometimes ambiguous semantics gives rise to further intricacies. The tool KITTeL [13] thus performs the termination analysis not on the source code level but on the level of a compiler intermediate representation (IR). This approach has the following advantages:

1. The IR is considerably simpler than real-life programming languages. This makes it possible to accept arbitrary (valid) programs as input.
2. The program whose termination behavior is analyzed is much closer to the program that is actually executed on the computer since most ambiguities of the semantics have already been resolved.
3. It becomes possible to analyze the termination behavior of programs written in any programming language that can be converted into the IR by a compiler front-end.

More concretely, KITTeL is based on the LLVM compiler framework and its intermediate language LLVM-IR [15]. Since there are compilers for various programming languages built atop of LLVM, KITTeL can be used for the termination analysis of programs written in C, C++, Objective-C, Ada, Fortran, etc. The main goal of KITTeL is the termination analysis of C programs.

A C program is first compiled into an LLVM-IR program using existing compiler front-ends such as llvm-gcc or clang. The LLVM-IR program is next converted into a transition system. This transition system is represented in the form of an *int-based term rewrite system (int-based TRS)* [12,13] in KITTeL, and the termination analysis itself is performed on this int-based TRS.[2] Fig. 1 gives an overview of the approach.

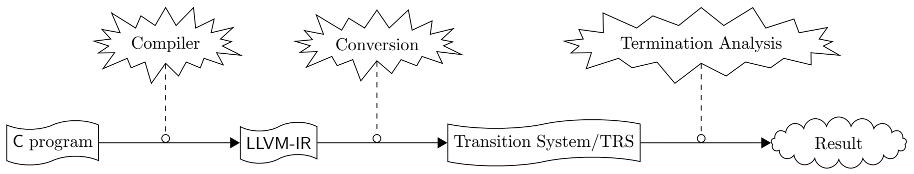


Fig. 1. Bird's-eye view of KITTeL's approach

2.1 LLVM-IR

An LLVM-IR program is an assembly program for a register machine with an unbounded number of registers. All registers in LLVM-IR are typed. Available types include a void type, integer types like i32 (where the bit-width is given explicitly but signed and unsigned types are not distinguished), floating point types, and derived types (such as pointer, array and structure types). The type i1 serves as a Boolean type. An LLVM-IR program consists of one or more functions. Each function is given as a list of basic blocks, where each basic block is a list of instructions. Execution of a function starts at the first basic block in the list.

LLVM-IR instructions can roughly be categorized into six classes:

1. *Three-address code (TAC)* instructions for arithmetical operations, comparison operations, and bitwise operations.
2. *Control flow* instructions: conditional and unconditional *branch* (br) instructions, *return* (ret) instructions, and *phi* (phi) instructions.
3. *Function calls* using call instructions.
4. *Memory access* instructions, namely load and store instructions.
5. *Address calculations* using getelementptr instructions.
6. *Auxiliary instructions* like type cast instructions (which do not change the bitlevel representation), signed and unsigned extension instructions, and truncation instructions.

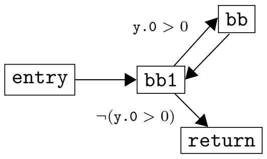
Branch instructions and return instructions only occur as the last instruction of a basic block and each basic block is terminated by one of these instructions.

[2] Alternatively, it would be possible to apply methods based on ranking functions [3,4,6,7,19], possibly combined with abstraction refinement [9,10,14,20], in order to investigate the termination behavior of the transition system.

A function gives rise to a *basic block graph* [22] which contains one node for each basic block and an edge from the basic block *bb* to the basic block *bb'* if *bb* is terminated by a branch instruction that can branch to *bb'*. For conditional branches, this edge is labeled by the condition under which the branch is taken.

Example 2. The following figure shows a C program, the LLVM-IR program obtained using the compiler front-end `llvm-gcc`, and the basic block graph of the function `power`:

```
int power(int x, int y) {
    int r = 1;
    while (y > 0) {
        r = r*x;
        y = y − 1;
    }
    return r;
}
```



```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

bb1:
  %y.0 = phi i32 [ %y, %entry ], [ %2, %bb ]
  %r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]
  %0 = icmp sgt i32 %y.0, 0
  br i1 %0, label %bb, label %return

bb:
  %1 = mul i32 %r.0, %x
  %2 = sub i32 %y.0, 1
  br label %bb1

return:
  ret i32 %r.0
}
```

This example is used in the next section in order to illustrate the conversion of an LLVM-IR program into a transition system/`int`-based TRS. ♢

LLVM-IR programs are in *static single assignment (SSA)* form, i.e., each register (variable) is assigned exactly once in the static program. This requires the use of phi-instructions, which may only occur at the beginning of basic blocks and select one of several values whenever the control flow in a program converges (e.g., after an `if-then-else` statement). Thus, the meaning of `%r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]` contained in the basic block `bb1` in Exa. 2 is that the register `%r.0` is assigned the value `1` if the control flow passed from `entry` to `bb1` and the value contained in `%1` if the control passed from `bb` to `bb1`.

3 Termination Analysis Using LLVM

For the termination analysis using LLVM, an LLVM-IR program is converted into a transition system whose termination behavior is then investigated. Within `KITTeL`, the transition system is represented in the form of an `int`-based TRS.

3.1 Translating LLVM-IR Programs into Transition Systems

For ease of exposition, it is assumed that the LLVM-IR program operates only on integer types, that there is exactly one function, and that this function does not contain any function calls. It thus only contains arithmetical and bitwise TAC instructions, comparison instructions, control flow instructions, and auxiliary instructions.[3]

For the translation into a transition system, each integer-typed function argument (i.e., of a type `i`k with $k > 1$) and each register defined by an integer-typed TAC instruction or phi-instruction is mapped to a variable. The set of these variables is denoted by $\mathcal{V}$. The transition system $(S, \Lambda, \to)$ corresponding to the LLVM-IR program is now constructed as follows:

– S contains one element for each node in the basic block graph.
– Λ denotes the set of transition constraints, which are quantifier-free constraints from (non-linear) integer arithmetic over the variables $\mathcal{V}$ and their primed versions in $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. As usual, the intended semantics of $\mathcal{V}'$ is to refer to the values of the variables in $\mathcal{V}$ after executing a transition.
– $\longrightarrow \subseteq S \times \Lambda \times S$ denotes the set of transitions: for each branch from the basic block *bb* to the basic block *bb'*, let $bb \xrightarrow{\lambda} bb'$ such that λ
  - describes the effect of the integer-typed TAC instructions in *bb*,
  - contains the condition under which the branch is taken, and
  - instantiates the variables corresponding to integer-typed phi-instructions in *bb'* according to their value if the control flow passes from *bb* to *bb'*.

The transition system is thus the basic block graph where each edge is, in addition to the branch condition, labeled by the state change caused by the transition.

In [13], bitvectors and bitvector arithmetic are abstracted to integers and integer arithmetic, respectively. Furthermore, there is no distinction between signed and unsigned operations. The arithmetical TAC instructions `add`, `sub`, and `mul` are replaced by the obvious integer arithmetic operations. The effect of the TAC instructions `sdiv`, `udiv`, `srem`, and `urem` is not modeled exactly. Instead, their result is abstracted to a fresh variable (optionally, constraints can restrict the range of this fresh variable, see [13] for details). The bitwise TAC instructions `and`, `or`, and `xor` are handled the same way. Finally, the extension and truncation instructions `sext`, `uext`, and `trunc` are modeled to not change the value of the integer.

Comparison instructions used in branch conditions are replaced in the obvious way by the corresponding integer comparisons. Signed and unsigned comparisons are not distinguished, i.e., `icmp ugt` and `icmp sgt` are both replaced by $>$. Boolean-typed TAC instructions used in branch conditions (`and`, `or`, and `xor`) are converted in the obvious way as well.

Example 3. For the LLVM-IR program from Exa. 2, the following transition system is obtained:

[3] Multiple (recursive) functions are handled by abstracting the return value of the called function by a fresh variable and by introducing suitable transitions to the entry state of the called function. Since `KITTeL` currently does not model the memory content, pointers are not tracked, `store` instructions are handled as no-ops, and `load` instructions are abstracted to fresh variables. Similarly, floating point instructions are handled as no-ops or abstracted to fresh variables.

$$\begin{tikzcd}
 & & \texttt{bb} \arrow[dl, bend left, "(3)"] \\
\texttt{entry} \arrow[r, "(1)"] & \texttt{bb1} \arrow[ur, "(2)"] \arrow[dr, "(4)"'] & \\
 & & \texttt{return}
\end{tikzcd}$$

(1) $\texttt{y.0}' \simeq \texttt{y} \wedge \texttt{r.0}' \simeq 1$
(2) $\texttt{y.0} > 0$
(3) $\texttt{y.0}' \simeq \texttt{y.0} - 1 \wedge \texttt{r.0}' \simeq \texttt{r.0} * \texttt{x} \wedge$
$\texttt{\%1}' \simeq \texttt{r.0} * \texttt{x} \wedge \texttt{\%2}' \simeq \texttt{y.0} - 1$
(4) $\neg(\texttt{y.0} > 0)$

Here, $\mathcal{V} = \{\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2}\}$. $\Diamond$

3.2 Representing Transition Systems Using int-Based TRSs

A transition system can be represented in the form of an `int`-based TRS [12,13]. The rewrite rules of an `int`-based TRS have the form $f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m)\ [\![\varphi]\!]$ where f and g are (uninterpreted) function symbols, $x_1, \ldots, x_n$ are pairwise distinct variables, $p_1, \ldots, p_m$ are (possibly non-linear) polynomials, and φ is a quantifier-free constraint from (non-linear) integer arithmetic that guards when a rewrite rule can be applied.[4] Then, each transition $s_1 \xrightarrow{\lambda} s_2$ gives rise to a rewrite rule

$$s_1(x_1, \ldots, x_n) \to s_2(e_1, \ldots, e_n)\ [\![\varphi]\!]$$

where $x_1, \ldots, x_n$ is a fixed order of the variables in $\mathcal{V}$ and

- $e_i = p$ if λ contains an "assignment" $x_i' \simeq p$ and $e_i = x_i$ otherwise.
- φ is λ with the "assignments" removed.

Example 4. Continuing Exa. 3,

$$\begin{aligned}
\texttt{entry}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2}) &\to \texttt{bb1}(\texttt{x}, \texttt{y}, \texttt{y}, 1, \texttt{\%1}, \texttt{\%2})\\
\texttt{bb1}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2}) &\to \texttt{bb}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2})\ [\![\texttt{y.0} > 0]\!]\\
\texttt{bb1}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2}) &\to \texttt{return}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2})\ [\![\neg(\texttt{y.0} > 0)]\!]\\
\texttt{bb}(\texttt{x}, \texttt{y}, \texttt{y.0}, \texttt{r.0}, \texttt{\%1}, \texttt{\%2}) &\to \texttt{bb1}(\texttt{x}, \texttt{y}, \texttt{y.0} - 1, \texttt{r.0}, \texttt{r.0} * \texttt{x}, \texttt{y.0} - 1)
\end{aligned}$$

is the representation of the transition system as an `int`-based TRS. $\Diamond$

When we talk about adding rewrite rules or replacing rewrite rules in the following, this can equally well be thought of as adding transitions or replacing transitions in the transition system (possibly adding new states as well if the rewrite rules introduce new function symbols).

The translation as outlined above produces rewrite rules where the function symbols have an unnecessarily large number of arguments. The number of arguments can be reduced by adapting standard compiler techniques:

1. *Backward slicing* with respect to the constraints removes all variables that are not relevant for the control flow of the program. In the running example, this removes the arguments corresponding to `x`, `r.0`, `%1`, and `%2`.

[4] In contrast to ordinary TRSs, $p_1, \ldots, p_m$ and φ may contain variables not occurring in $x_1, \ldots, x_n$.

2. *Liveness analysis* removes variables before they are defined or after they are no longer needed. In the running examples, this removes the argument corresponding to `y.0` from the function symbols `entry` and `return` and the argument corresponding to `y` from all function symbols but `entry`.

Example 5. Applying these methods,

$$\begin{aligned} \texttt{entry}(\texttt{y}) &\to \texttt{bb1}(\texttt{y}) && (1)\\ \texttt{bb1}(\texttt{y.0}) &\to \texttt{bb}(\texttt{y.0})\ [\![\texttt{y.0} > 0]\!] && (2)\\ \texttt{bb1}(\texttt{y.0}) &\to \texttt{return}()\ [\![\neg(\texttt{y.0} > 0)]\!] && (3)\\ \texttt{bb}(\texttt{y.0}) &\to \texttt{bb1}(\texttt{y.0} - 1) && (4) \end{aligned}$$

is the final `int`-based TRS obtained from Exa. 4. ♢

3.3 Termination Analysis of `int`-Based TRSs

The `int`-based TRS obtained from an LLVM-IR program is then analyzed for termination using term rewriting techniques. The key techniques are:

1. Determining which rules may be applied following each other and a decomposition into non-trivial strongly connected components (SCCs). This step roughly corresponds to a decomposition into loops of the program.
2. Automatically generating well-founded relations based on (possibly non-linear) polynomial interpretations. Here, a polynomial interpretation maps the function symbols to polynomials such that a function symbol f with n arguments is mapped to a polynomial $\mathcal{P}ol(f) \in \mathbb{Z}[X_1, \ldots, X_n]$. A polynomial interpretation is applied to terms by applying the polynomial assigned to the function symbol to the arguments, i.e., by letting $\mathcal{P}ol(f(p_1, \ldots, p_n)) = \mathcal{P}ol(f)(p_1, \ldots, p_n)$. A polynomial interpretation gives rise to a well-founded relation by letting (for terms s, t and a quantifier-free integer constraint φ)

$$s \succ_{\mathcal{P}ol} t\ [\![\varphi]\!] \quad \text{iff} \quad \varphi \Rightarrow \mathcal{P}ol(s) > \mathcal{P}ol(t) \text{ and } \varphi \Rightarrow \mathcal{P}ol(s) \geq 0 \text{ are valid}$$

   Similarly, $s \succsim_{\mathcal{P}ol} t\ [\![\varphi]\!]$ iff $\varphi \Rightarrow \mathcal{P}ol(s) \geq \mathcal{P}ol(t)$ is valid (these relations are in general undecidable due to non-linearity). Then, all rules $s \to t\ [\![\varphi]\!]$ with $s \succ_{\mathcal{P}ol} t\ [\![\varphi]\!]$ can be removed from an `int`-based TRS if all remaining rules $s' \to t'\ [\![\varphi']\!]$ satisfy $s' \succsim_{\mathcal{P}ol} t'\ [\![\varphi']\!]$. Methods for the automatic generation of suitable polynomial interpretations are discussed in [12,13]. The current implementation is restricted to polynomials of the form $\sum_{i=1}^{n} a_i X_i + \sum_{j=1}^{n} b_j X_j^2 + c$, with $a_i, b_j, c \in \mathbb{Z}$. Notice that this is more general than the linear ranking functions that are considered in [8] since the coefficients are not restricted and a limited form of non-linearity is supported.
3. Combining rewrite rules that may be applied following each other into new rewrite rules. This corresponds to combining transitions that may be applied successively in the transition system.

These techniques can be applied modularly in any order and combination. Concretely, `KITTeL` applies them in a loop, where each loop iteration applies the first technique from the list that is successfully applicable. For SCC decomposition and the generation of polynomial interpretations, SMT-solvers such as `Yices` [11] or `Z3` [16] are used for solving (linear) integer arithmetic constraints, where `KITTeL` uses `Yices` by default. Details are discussed in [12,13].

Example 6. Termination of the `int`-based TRS from Exa. 5 is easily established using these techniques, thus establishing termination of the C program using integer arithmetic. First, SCC decomposition removes the rewrite rules (1) and (3). Next, the polynomial interpretation $\mathcal{P}ol(\texttt{bb1}) = X_1$, $\mathcal{P}ol(\texttt{bb}) = X_1 - 1$ removes the rewrite rule (2) since $\texttt{bb1}(\texttt{y.0}) \succ_{\mathcal{P}ol} \texttt{bb}(\texttt{y.0})\ [\![\texttt{y.0} > 0]\!]$ and $\texttt{bb}(\texttt{y.0}) \succsim_{\mathcal{P}ol} \texttt{bb1}(\texttt{y.0} - 1)$. Finally, SCC decomposition finishes the termination proof. ♢

4 Encoding Bitvector Arithmetic

In order to model the bitvector semantics of machine arithmetic, bitvectors can be represented by (unbounded) integers if the wrap-around behavior of the arithmetical operations is modeled properly. There are two natural choices for the representation of a bitvector by an integer:

1. The bitvector $b_{n-1} \cdots b_0$ is represented by its *unsigned* value $\sum_{i=0}^{n-1} b_i \cdot 2^i \in \{0, \ldots, 2^n - 1\}$.
2. The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) value $-b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$.

In the following, the representation by the signed value is considered (the implementation in `KITTeL` supports both possibilities).

The semantics of the bitvector operations and the wrap-around behavior of bitvector arithmetic is handled by a two-phase approach:

1. For the generation of the transition system/`int`-based TRS, the conversion of comparison instructions, arithmetical and bitwise TAC instructions, and extension/truncation instructions is adapted in order to correspond to their semantics on bitvectors. In particular, signed and unsigned operations are carefully distinguished. The wrap-around behavior of the arithmetical operations is not yet taken into account since this will be done afterwards in the second phase.
2. The wrap-around behavior of the arithmetical instructions is modeled by explicitly normalizing the results of arithmetical operations in order to ensure that they are within the appropriate ranges.

In the following, $\|x\|$ for a variable x denotes the size of the bitvector of the LLVM-IR value corresponding to x. This extends to arithmetical expressions in the obvious way. Furthermore, $\mathsf{intmin}(n) = -2^{n-1}$ and $\mathsf{intmax}(n) = 2^{n-1} - 1$ denote the minimal and maximal signed value that can be represented by bitvectors of size n.

4.1 Phase 1

In contrast to Sect. 3, signed and unsigned comparison instructions are now distinguished. Since a bitvector is represented by its signed value, the signed comparison instructions are still converted to the corresponding integer comparisons and only the unsigned comparison instructions need to be handled differently.

`icmp eq i`k x`,` y	$x \simeq y$	`icmp ne i`k x`,` y	$x \not\simeq y$
`icmp ugt i`k x`,` y	$\mathsf{ugt}(x, y)$	`icmp sgt i`k x`,` y	$x > y$
`icmp uge i`k x`,` y	$\mathsf{ugt}(x, y) \vee x \simeq y$	`icmp sge i`k x`,` y	$x \geq y$
`icmp ult i`k x`,` y	$\mathsf{ult}(x, y)$	`icmp slt i`k x`,` y	$x < y$
`icmp ule i`k x`,` y	$\mathsf{ult}(x, y) \vee x \simeq y$	`icmp sle i`k x`,` y	$x < y$

Here, $\mathsf{ugt}(x, y)$ and $\mathsf{ult}(x, y)$ encode unsigned greater-than and less-than comparisons on bitvectors, respectively, where bitvectors are represented by their signed value. They are defined as follows:

$$\begin{aligned}\mathsf{ugt}(x, y) = &\quad (x \geq 0 \wedge y \geq 0 \wedge x > y)\\ &\vee\ (x \geq 0 \wedge y < 0)\\ &\vee\ (x < 0 \wedge y < 0 \wedge x > y)\\ \mathsf{ult}(x, y) = &\ \mathsf{ugt}(y, x)\end{aligned}$$

Arithmetical and bitwise TAC instructions as well as auxiliary instructions are converted as follows. Again, signed and unsigned operations are carefully distinguished. The results of `sdiv`, `udiv`, `srem`, `urem`, `and`, and `or` instructions are represented by a fresh variable n_i. Furthermore, constraints on the value of this variable are added. These constraints describe the (approximated) semantics of the corresponding instructions on bitvectors.

arithmetic	$z =$ `add i`k x`,` y	$z' \simeq x + y$
	$z =$ `sub i`k x`,` y	$z' \simeq x - y$
	$z =$ `mul i`k x`,` y	$z' \simeq x * y$
	$z =$ `sdiv i`k x`,` y	$z' \simeq n_i \wedge \mathsf{sdiv}(x, y, n_i)$
	$z =$ `udiv i`k x`,` y	$z' \simeq n_i \wedge \mathsf{udiv}(x, y, n_i)$
	$z =$ `srem i`k x`,` y	$z' \simeq n_i \wedge \mathsf{srem}(x, y, n_i)$
	$z =$ `urem i`k x`,` y	$z' \simeq n_i \wedge \mathsf{urem}(x, y, n_i)$
bitwise operations	$z =$ `and i`k x`,` y	$z' \simeq n_i \wedge \mathsf{and}(x, y, n_i)$
	$z =$ `or i`k x`,` y	$z' \simeq n_i \wedge \mathsf{or}(x, y, n_i)$
	$z =$ `xor i`k x`,` y	$z' \simeq n_i$
extension/truncation	$z =$ `sext i`k x `to i`l	$z' \simeq n_i$
	$z =$ `uext i`k x `to i`l	$z' \simeq x \wedge \mathsf{uext}(z, n_i)$
	$z =$ `trunc i`k x `to i`l	$z' \simeq x$

The formulas for the constraints typically perform a case distinction on the inputs x and y in order to obtain constraints on the result n_i. For instance for and, if both x and y are positive (most significant bit is zero), then n_i is positive as well and exceeds neither x nor y. If exactly one of x and y is positive, then

n_i is positive and does not exceed the positive input. Finally, if both x and y are negative (most significant bit is one), then n_i is negative as well and again exceeds neither x nor y. The formulas are as follows:

$$\begin{aligned}
\mathsf{sdiv}(x,y,n_i) = {} & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq x) \vee (y \simeq -1 \wedge n_i \simeq -x) \\
& \vee (y > 1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < x) \\
& \vee (y > 1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > x) \\
& \vee (y < -1 \wedge x > 0 \wedge n_i \leq 0 \wedge n_i > -x) \\
& \vee (y < -1 \wedge x < 0 \wedge n_i \geq 0 \wedge n_i < -x) \\
\mathsf{udiv}(x,y,n_i) = {} & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq x) \\
& \vee (\mathsf{ugt}(y,1) \wedge \mathsf{ugt}(x,0) \wedge \mathsf{ugt}(n_i,0) \wedge \mathsf{ult}(n_i,x)) \\
\mathsf{srem}(x,y,n_i) = {} & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq 0) \vee (y \simeq -1 \wedge n_i \simeq 0) \\
& \vee (y > 1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < y) \\
& \vee (y > 1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > -y) \\
& \vee (y < -1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < y) \\
& \vee (y < -1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > y) \\
\mathsf{urem}(x,y,n_i) = {} & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq 0) \\
& \vee (\mathsf{ugt}(y,1) \wedge \mathsf{ugt}(x,0) \wedge \mathsf{ult}(n_i,y)) \\
\mathsf{and}(x,y,n_i) = {} & (x \geq 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq x \wedge n_i \leq y) \\
& \vee (x \geq 0 \wedge y < 0 \wedge n_i \geq 0 \wedge n_i \leq x) \\
& \vee (x < 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq y) \\
& \vee (x < 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \leq x \wedge n_i \leq y) \\
\mathsf{or}(x,y,n_i) = {} & (x \geq 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \geq x \wedge n_i \geq y) \\
& \vee (x \geq 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \geq y) \\
& \vee (x < 0 \wedge y \geq 0 \wedge n_i < 0 \wedge n_i \geq x) \\
& \vee (x < 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \geq x \wedge n_i \geq y) \\
\mathsf{uext}(x,n_i) = {} & (x \geq 0 \wedge n_i \simeq x) \vee \left(x < 0 \wedge n_i \simeq x + 2^{\|x\|}\right)
\end{aligned}$$

Example 7. Recall the function `f` from Exa. [1]:

```
void f(int i) {
    while (i > 0) {
        ++i;
    }
}
```

```
define void @f(i32 %i) {
entry:
  br label %bb1

  bb1:
    %i.0 = phi i32 [ %i, %entry ], [ %1, %bb ]
    %0 = icmp sgt i32 %i.0, 0
    br i1 %0, label %bb, label %return

  bb:
    %1 = add nsw i32 %i.0, 1
    br label %bb1

  return:
    ret void
}
```

$$\begin{aligned}
&\mathtt{entry(i)} \to \mathtt{bb1(i)} \\
&\mathtt{bb1(i.0)} \to \mathtt{bb(i.0)}\ [\![\mathtt{i.0} > 0]\!] \\
&\mathtt{bb1(i.0)} \to \mathtt{return()}\ [\![\neg(\mathtt{i.0} > 0)]\!] \\
&\mathtt{bb(i.0)} \to \mathtt{bb1(i.0}+1)
\end{aligned}$$

After phase 1 and the slicing/liveness analysis based elimination of arguments, the `int`-based TRS show above is obtained. Notice that the wrap-around behavior caused by an overflow in $\mathtt{i.0} + 1$ it not yet taken care of. This is addressed in phase 2. $\Diamond$

4.2 Phase 2

Next, the wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS. For this, the wrap-around behavior is simulated by explicitly normalizing the resulting integers to be in the appropriate ranges. For a variable x, the normalization consists of the repeated addition or subtraction of the correction $2^{\|x\|}$ until x is in the range of bitvectors of size $\|x\|$. The following construction is applied separately to each rewrite rule.

No arithmetical operations in the constraint: At first, it is assumed that the constraint of the rewrite rule does not contain any arithmetical operations. The general case is discussed subsequently.

Then, the rewrite rule

$$\rho : f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m)\ [\![\varphi]\!]$$

is replaced by the following rewrite rules (notice that $p_1, \ldots, p_m$ may be outside of the appropriate ranges even if the instantiations of all variables are within their appropriate ranges):

$$\begin{aligned}
f(x_1, \ldots, x_n) &\to g^\sharp(p_1, \ldots, p_m)\ [\![\varphi \wedge \mathsf{inrange}(\mathcal{V}(\rho))]\!]\\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 + 2^{\|x_1\|}, \ldots, x_m)\ [\![x_1 < \mathsf{intmin}(\|x_1\|)]\!]\\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 - 2^{\|x_1\|}, \ldots, x_m)\ [\![x_1 > \mathsf{intmax}(\|x_1\|)]\!]\\
&\ \ \vdots\\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m + 2^{\|x_m\|})\ [\![x_m < \mathsf{intmin}(\|x_m\|)]\!]\\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m - 2^{\|x_m\|})\ [\![x_m > \mathsf{intmax}(\|x_m\|)]\!]\\
g^\sharp(x_1, \ldots, x_m) &\to g(x_1, \ldots, x_m)\ [\![\mathsf{inrange}(\{x_1, \ldots, x_m\})]\!]
\end{aligned}$$

Here, $g^\sharp$ is a fresh function symbol, $\mathcal{V}(\rho)$ are the variables occurring in ρ, and

$$\mathsf{inrange}(V) = \bigwedge_{v \in V} (v \geq \mathsf{intmin}(\|v\|) \wedge v \leq \mathsf{intmax}(\|v\|))$$

expresses that all arithmetical expressions in V are in the appropriate ranges.[5]

[5] It of course suffices to add the $g^\sharp(\ldots) \to g^\sharp(\ldots)\ [\![\ldots]\!]$ rules only for those x_i where p_i contains arithmetical operations. If there is no such i, then the rewrite rule ρ can be taken as is.

Example 8. Continuing Exa. 7, phase 2 produces

$$\texttt{entry}(\texttt{i}) \to \texttt{bb1}(\texttt{i})\ [\![\mathsf{inrange}(\{\texttt{i}\})]\!] \tag{5}$$

$$\texttt{bb1}(\texttt{i.0}) \to \texttt{bb}(\texttt{i.0})\ [\![\texttt{i.0} > 0 \wedge \mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{6}$$

$$\texttt{bb1}(\texttt{i.0}) \to \texttt{return}()\ [\![\neg(\texttt{i.0} > 0) \wedge \mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{7}$$

$$\texttt{bb}(\texttt{i.0}) \to \texttt{bb1}^\sharp(\texttt{i.0} + 1)\ [\![\mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{8}$$

$$\texttt{bb1}^\sharp(\texttt{i.0}) \to \texttt{bb1}^\sharp(\texttt{i.0} + 2^{32})\ [\![\texttt{i.0} < \mathsf{intmin}(32)]\!] \tag{9}$$

$$\texttt{bb1}^\sharp(\texttt{i.0}) \to \texttt{bb1}^\sharp(\texttt{i.0} - 2^{32})\ [\![\texttt{i.0} > \mathsf{intmax}(32)]\!] \tag{10}$$

$$\texttt{bb1}^\sharp(\texttt{i.0}) \to \texttt{bb1}(\texttt{i.0})\ [\![\mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{11}$$

as an `int`-based TRS. ◊

Arithmetical operations in the constraint: If the constraint φ contains arithmetical operations, then the results of these operations need to be normalized before the constraint can be evaluated. This can be done by adding "dummy" variables for these arithmetic expressions. If φ contains the maximal arithmetic expressions $q_1, \ldots, q_l$, then the rewrite rule ρ is first converted into

$$\begin{aligned}
f(x_1,\ldots,x_n) &\to f^\dagger(x_1,\ldots,x_n,q_1,\ldots,q_l)\ [\![\mathsf{inrange}(\mathcal{V}(\rho))]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1+2^{\|y_1\|},\ldots,y_l)\ [\![y_1 < \mathsf{intmin}(\|y_1\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1-2^{\|y_1\|},\ldots,y_l)\ [\![y_1 > \mathsf{intmax}(\|y_1\|)]\!]\\
&\ \vdots\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l+2^{\|y_l\|})\ [\![y_l < \mathsf{intmin}(\|y_l\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l-2^{\|y_l\|})\ [\![y_l > \mathsf{intmax}(\|y_l\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to g(p_1,\ldots,p_m)\ [\![\widehat{\varphi} \wedge \mathsf{inrange}(\mathcal{V}(\rho)\cup\{y_1,\ldots,y_l\})]\!]
\end{aligned}$$

were $f^\dagger$ is a fresh function symbol and $\widehat{\varphi}$ is obtained from φ by replacing q_i by y_i for all $1 \le i \le l$. Next, the last newly generated rewrite rule is converted as in the previous paragraph.

Optimizations: Notice that the $g^\sharp$-rules (and the $f^\dagger$-rules) essentially use loops for the normalization. Often, a small upper bound on the number of needed loop iterations is known. For instance in $\texttt{i.0} + 1$ from Exa. 8, the correction $2^{\|\texttt{i.0}\|}$ needs to be applied at most once. Thus, the loop for the variable corresponding to $\texttt{i.0} + 1$ can be eliminated by applying the correction zero or one times. Furthermore, $\texttt{i.0}+1$ can only overflow but never underflow, i.e., an addition of the correction $2^{\|\texttt{i.0}\|}$ does not need to be considered at all.

Example 9. Continuing Exa. 8, the rewrite rules (8)–(11) can be replaced by rewrite rules obtained by combining rewrite rules (8) and (11) and rewrite rules

(8), (10), and (11) into new rewrite rules (these are all possible ways to execute the normalization loop zero or one times for a possible overflow). Then,

$$\begin{aligned}
\mathtt{entry}(\mathtt{i}) &\to \mathtt{bb1}(\mathtt{i})\ [\![\mathsf{inrange}(\{\mathtt{i}\})]\!]\\
\mathtt{bb1}(\mathtt{i.0}) &\to \mathtt{bb}(\mathtt{i.0})\ [\![\mathtt{i.0} > 0 \land \mathsf{inrange}(\{\mathtt{i.0}\})]\!]\\
\mathtt{bb1}(\mathtt{i.0}) &\to \mathtt{return}()\ [\![\neg(\mathtt{i.0} > 0) \land \mathsf{inrange}(\{\mathtt{i.0}\})]\!]\\
\mathtt{bb}(\mathtt{i.0}) &\to \mathtt{bb1}(\mathtt{i.0}+1)\ [\![\mathsf{inrange}(\{\mathtt{i.0}\}) \land \mathsf{inrange}(\{\mathtt{i.0}+1\})]\!]\\
\mathtt{bb}(\mathtt{i.0}) &\to \mathtt{bb1}(\mathtt{i.0}+1-2^{32})\ [\![\mathsf{inrange}(\{\mathtt{i.0}\}) \land \mathtt{i.0}+1 > \mathsf{intmax}(32)\\
&\qquad\qquad \land \mathsf{inrange}(\{\mathtt{i.0}+1-2^{32}\})]\!]
\end{aligned}$$

is obtained. Termination of this `int`-based TRS (or the `int`-based TRS from Exa. 8) is easily established using the techniques from Sect. 3.3, thus establishing termination of the C program using bitvector arithmetic. ◊

5 Experimental Results and Evaluation

In order to assess the practicality of the proposed method, we have implemented it in the termination prover `KITTeL` [13]. For an empirical evaluation, we ran `KITTeL` on the 61 examples presented in [8, Sect. 4.2].[6] These examples were extracted from the *Windows Driver Development Kit*. Of these 61 examples, 51 are terminating and 10 are nonterminating using bitvector arithmetic. Since [8] also provides experimental results for an implementation of their methods, a direct comparison is easily possible.[7] Furthermore, these examples were also used as benchmarks in [26] in order to evaluate an SMT solver.[8] These results are contained in the evaluation as well.

All tools were (assumed to be) run with a timeout of 60s for each example. A summary of the results is given in the following table.[9] In this table, "yes" means a successful termination proof, "maybe" means that execution stopped before the timeout without producing a successful termination proof, and "timeout" means that the timeout was reached. The calculation of average times only takes "yes" and "maybe" answers into account, whereas the average "yes" time only takes "yes" answers into account.

[6] These are all examples considered in [8] for which the source code is available at `http://www.cprover.org/termination/ranking/index.shtml`.

[7] Due to different machines used for the experiments ([8] uses an 8-core Intel® Xeon® 3GHz with 16GB of RAM, whereas our experiments where performed on a 2-core Intel® Core$^{\mathrm{TM}}$ 2 Duo 2.4GHz with 4GB of RAM), the comparison of the runtimes is not completely accurate.

[8] A "sat" in [26] means that termination could be shown whereas an "unsat" means that termination could not be shown.

[9] Complete results for `KITTeL` can be found at `http://baldur.iti.kit.edu/~falke/kittel-bitvector/`

	KITTeL	[8], quantifier elimination	[8], SAT solver	[8], QBF solver	[8], SMT solver [26]
# yes	38	30	34	8	27
# maybe	8	24	26	11	14
# timeout	15	7	1	42	20
average time	6.6s	10.8s	2.2s	7.1s	11.8s
average "yes" time	3.3s	11.1s	2.7s	13.4s	12.2s

To summarize the results, KITTeL is more successful than the most successful method from [8] (even in combination with [26]) and needs a comparable average "yes" time. The higher number of timeouts and the higher average time of KITTeL is due to the modular analysis loop used for the termination analysis of int-based TRSs within KITTeL. Whereas the methods from [8] give up as soon as they encounter a path for which they cannot find a ranking function, KITTeL can usually continue its analysis loop by combining rewrite rules (in the evaluation, the number of applications of this technique was restricted to 15).

6 Conclusions

We have shown how the method presented in [13] can be extended to show termination of programs using bitvector arithmetic. In particular, the wrap-around behavior caused by under- and overflows is correctly modeled. Bitwise operations such as and and or, as well as arithmetical operations such as division and remainder, are soundly approximated. This way, unsound and incomplete abstractions caused by identifying bitvectors with integers and bitvector arithmetic with integer arithmetic (as is done by nearly all current methods for proving termination of imperative programs) are avoided and the behavior of an imperative program is modeled in the way it is executed on the computer.

An implementation of the proposed method in the tool KITTeL [13] has been evaluated on 61 examples extracted from the *Windows Driver Development Kit* that were also used in [8]. The performance of KITTeL on these examples is very encouraging—better than the performance of the tool presented in [8]. In future work, we plan to extend KITTeL in order to reason about termination due to the traversal of well-formed linked data structures on the heap. Furthermore, we plan to explore refined sound approximations for arithmetic and bitwise operations. Finally, since the modeling of bitvector arithmetic adds overhead in comparison to integer arithmetic, the development of a CEGAR-like approach which starts using integer arithmetic and gradually refines the model to use bitvector arithmetic is a promising direction for future research.

Acknowledgments. We thank the anonymous reviewers for helpful comments and for pointing us to the evaluation contained in [26].

References

1. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
2. Babić, D., Musuvathi, M.: Modular arithmetic decision procedure. Tech. Rep. TR-2005-114, Microsoft Research Redmond (2005)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear Ranking with Reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination Analysis of Integer Linear Loops. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 488–502. Springer, Heidelberg (2005)
5. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java bytecode programs by term rewriting. In: RTA 2011(2011)
6. Colón, M., Sipma, H.: Synthesis of Linear Ranking Functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
7. Colón, M., Sipma, H.: Practical Methods for Proving Program Termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
8. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking Function Synthesis for Bit-Vector Relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
9. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006, pp. 415–426 (2006)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Falke, S., Kapur, D.: A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
13. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: RTA 2011, pp. 41–50 (2011)
14. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
15. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88 (2004)
16. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM TOPLAS 29(5), 29:1–29:27 (2007)
18. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: RTA 2010, pp. 259–276 (2010)
19. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)

20. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004, pp. 32–41 (2004)
21. Simon, A., King, A.: Taming the Wrapping of Integer Arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)
22. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: SSV 2010 (2010)
23. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM TOPLAS 32(3), 8:1–8:70 (2010)
24. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)
25. Wang, B.Y.: On the Satisfiability of Modular Arithmetic Formulae. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 186–199. Springer, Heidelberg (2006)
26. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: FMCAD 2010, pp. 239–246 (2010)

Specifying and Verifying the Correctness of Dynamic Software Updates

Christopher M. Hayden[1], Stephen Magill[1], Michael Hicks[1], Nate Foster[2], and Jeffrey S. Foster[1]

[1] Computer Science Department, University of Maryland, College Park
{hayden,smagill,mwh,jfoster}@cs.umd.edu
[2] Computer Science Department, Cornell University
jnfoster@cs.cornell.edu

Abstract. Dynamic software updating (DSU) systems allow running programs to be patched on-the-fly to add features or fix bugs. While dynamic updates can be tricky to write, techniques for establishing their correctness have received little attention. In this paper, we present the first methodology for automatically verifying the correctness of dynamic updates. Programmers express the desired properties of an updated execution using *client-oriented specifications* (CO-specs), which can describe a wide range of client-visible behaviors. We verify CO-specs automatically by using off-the-shelf tools to analyze a *merged* program, which is a combination of the old and new versions of a program. We formalize the merging transformation and prove it correct. We have implemented a program merger for C, and applied it to updates for the Redis key-value store and several synthetic programs. Using Thor, a verification tool, we could verify many of the synthetic programs; using Otter, a symbolic executor, we could analyze every program, often in less than a minute. Both tools were able to detect faulty patches and incurred only a factor-of-four slowdown, on average, compared to single version programs.

1 Introduction

Dynamic software updating (DSU) systems allow programs to be patched on-the-fly, to add features or fix bugs without incurring downtime. DSU systems were originally developed for a few limited domains such as telecommunications networks, financial transaction processors, and the like, but are now becoming pervasive. Ksplice, recently acquired by Oracle, supports applying Linux kernel security patches dynamically [16]. The Erlang language, which provides built-in support for dynamic updates, is gaining in popularity for building server programs [2]. Many web applications employ DSU techniques to provide 24/7 service to a global audience—for these systems, there is no single time of day when taking down the service to perform upgrades is acceptable.

Given the increasing need for DSU, a natural question is: How can developers ensure a dynamically updated program will behave correctly? Today, developers need to reason manually about all the pieces of an updating program: the old program version, the new program version, and code that transforms the state of the (old) running version into the form expected by the new version. Moreover, they need to repeat this reasoning process for each allowable "update point" during execution. In our experience this is a tricky proposition in which it is all too easy to make mistakes. Despite such difficulties, most DSU systems do not address the issue of correctness, or they focus exclusively on generic safety properties, such as type safety, that rule out obviously wrong behavior [7,23,24,25] but are insufficient for establishing correctness [12].

This paper presents a methodology for verifying the correctness of dynamic updates. Rather than propose a new verification algorithm that accounts for the semantics of updating, we develop a novel program transformation that produces a program suitable for verification with off-the-shelf tools. Our transformation *merges* an old program and an update into a program that simulates running the program and applying the update at any allowable point. We formalize our transformation and prove that it is correct (Section 3).

We are particularly interested in using our transformation to prove execution properties from clients' points of view, to show that a dynamic update does not disrupt active sessions. For example, suppose we wish to update a key-value store such as Redis [21] so that it uses a different internal data structure. To verify this update's transformation code, we could prove that values inserted into the store by the client are still present after it is dynamically updated. We call such specifications *client-oriented specifications* (or *CO-specs* for short).

We have identified three categories of CO-specs that capture most properties of interest: *backward-compatible* CO-specs describe properties that are identical in the old and new versions; *post-update* CO-specs describe properties that hold after new features are added or bugs are fixed by an update; and *conformable* CO-specs describe properties that are identical in the old and new versions, modulo uniform changes to the external interface. CO-specs in these categories can often be mechanically constructed from CO-specs written for either the old or new program alone. Thus, if a programmer is inclined to verify each program version using CO-specs, there is little additional work to verify a dynamic update between the two. Nevertheless, some interesting and subtle properties lie outside these categories, so our framework also allows arbitrary properties to be expressed (Section 2).

We have implemented our merging transformation for C programs and used it in combination with two existing tools to verify properties of several dynamic updates (Section 4). We chose the symbolic executor Otter [22] and the verification tool Thor [17] as they represent two ends of the design space: symbolic execution is easy to use and scales reasonably well but is incomplete, while verification scales less well but provides greater assurance. We wrote two synthetic benchmarks, a key-value store and a multiset implementation, and designed dynamic patches for them based on realistic changes (e.g., one change was inspired by an update to the storage server Cassandra [5]). We also wrote dynamic patches for six releases of Redis [21], a popular, open-source key-value store. We used the Redis code as is, and wrote the state transformation code ourselves.

We checked all the benchmark programs with Otter and verified several properties of the synthetic updates using Thor. Both tools successfully uncovered bugs that were intentionally and unintentionally introduced in the state transformation code. The running time for verification of merged programs was roughly four times slower than single-version checking. This slowdown was due to the additional branching introduced by update points and the need to analyze the state transformer code. As tools become faster and more effective, our approach will scale with them. In summary, this paper makes three main contributions:

– It presents the first automated technique for verifying the behavioral correctness of dynamic updates.
– It proposes *client-oriented specifications* as a means to specify general update correctness properties.
– It shows the effectiveness of merging-based verification on practical examples, including Redis [21], a widely deployed server program.

2 Defining Dynamic Software Update Correctness

Before we can set out verifying DSU correctness, we have to decide what correctness is. In this section, we first review previously proposed notions of correctness and argue why they are insufficient for our purposes. Then we propose *client-oriented specifications* (CO-specs) as a means of specifying correctness properties, and argue that this notion overcomes limitations of prior notions. We also describe a simple refactoring that allows CO-specs to be used to verify client-server programs that communicate over a network.

2.1 Prior Work on Update Correctness

Kramer and Magee [15] proposed that updates are correct if they are *observationally equivalent*—i.e., if the updated program preserves all observable behaviors of the old program. Bloom and Day [3] observed that, while intuitive, this is too restrictive: an update may fix bugs or add new features.

To address the limitations of strict observational equivalence, Gupta et al. [9] proposed *reachability.* This condition classifies an update as correct if, after the update is applied, the program eventually *reaches* some state of the new program. Reachability thus admits bugfixes, where the new state consists of the corrected code and data, as well as feature additions, where the new state is the old data plus the new code and any new data. Unfortunately, reachability is both too permissive and too restrictive, as shown by the following example. Version 1.1.2 of the vsftpd FTP server introduced a feature that limits the number of connections from a single host. If we update a running vsftpd server, we would expect it to preserve any active connections. But doing so violates reachability. If the number of connections from a particular host exceeds the limit and these connections remain open indefinitely, the server will never enter a reachable state of the new program. On the other hand, reachability would allow an update that terminates all existing connections. This is almost certainly not what we want—if we were willing to drop existing connections we could just restart the server!

We believe that the flaw in all of these approaches is that they attempt to define correctness in a completely general way. We think it makes more sense for programmers to specify the behavior they expect as a collection of properties. Some properties will apply to multiple versions of the program while other properties will change as the program evolves. Because the goal of a dynamic update is to preserve active processing and state, the properties should express the expected continuity that a dynamic update is meant to provide to active clients. We therefore introduce *client-oriented specifications* (CO-specs) to specify update properties that satisfy these requirements.

2.2 Client-Oriented Specifications

We can think of a CO-spec as a kind of client program that opens connections, sends messages, and asserts that the output received is correct. CO-specs resemble tests, but certain elements of the test code are left abstract for generality (cf. Figure 1). For example, consider again reasoning about updates to a key-value store such as Redis. A CO-spec might model a client that inserts a key-value pair into the store and then looks up the key, checking that it maps to the correct value (even if a dynamic update has occurred in the meantime). We can make such a CO-spec general by leaving certain elements like the particular keys or values used unconstrained. Similarly, we can allow arbitrary actions to be interleaved between the insert and lookup. Such specifications capture essentially arbitrary client interactions with the server.

Our goal is to use our program transformation, defined in Section 3, to produce a *merged* program that we can verify using off-the-shelf tools. But existing tools only verify single programs in isolation, so we cannot literally write CO-specs as client programs that communicate with a server being updated. To verify a CO-spec in a real client-server program we replace the server's `main` function the CO-spec and call the relevant server functions directly. In doing so, we are checking the server's core functionality, but not its main loop or any networking code. For example, suppose our key-value store implements functions `get` and `set` to read and write mappings from the store, and the server's main loop would normally dispatch to these functions. CO-specs would call the functions directly as shown in Figure 1. Here, ? denotes a non-deterministically chosen (integer) value, and `assume` and `assert` have their standard semantics. If updates are permitted while executing either `get` or `set`, verifying Figure 1(b) will establish that the assertions at the end of the specification hold no matter when the update takes place.

In our experience writing CO-specs for updates, we have found that they often fall into one of the following categories:

- *Backward-compatible CO-specs* describe behaviors that are unaffected by an update. For the data structure-changing update to Redis mentioned earlier, the CO-spec in Figure 1(b) would check that existing mappings are preserved.
- *Post-update CO-specs* describe behavior specific to the new program version. For example, suppose we added a `delete` feature to the key-value store. Then

```
int get(int k, int *v);
void set(int k, int v);

void arbitrary (int k1) {
 int k2 = ?, v = ?;
 if (k1 == k2 || ?)
   get(k2,&v);
 else set (k2,v);
}
```

(a) interface, helper

```
void back_compat_spec() {
  int k = ?, v_in = ?;
  int v_out, found;
  set(k, v_in );
  while(?) arbitrary (k);
  found = get(k,&v_out);
   assert (found &&
           v_out == v_in);
}
```

(b) backward-compat. spec

```
void post_update_spec() {
  int k = ?;
  int v_out, found;
  while(?) arbitrary (?);
  assume(is_updated);
  delete (k);
  found = get(k,&v_out);
  assert (!found);
}
```

(c) post-update spec

Fig. 1. Sample C specifications for key-value store

the CO-spec in Figure 1(c) verifies that, after the update, the feature is working properly. The CO-spec employs the flag is_updated, which is true after an update has taken place, to ensure that we are testing the new or changed functionality after the update. We discuss the semantics of this flag in Section 3.

– *Conformable CO-specs* describe updates that change interfaces, but preserve core functionality. For example, suppose we added namespaces to our key-value store, so that get and set take an additional namespace argument. The state transformation code would map existing entries to a default namespace. A conformable CO-spec could check that mappings inserted prior to the update are present in the default namespace afterward; in essence, the CO-spec would associate old-version calls with new-version calls at the default namespace. (Further details are given in our technical report [11].)

These categories encompass prior notions of correctness. Backward compatible specifications capture the spirit of Kramer and Magee's condition, but apply to individual, not all, behaviors. The combination of backward-compatible and post-update specifications capture Bloom and Day's notions of "future-only implementations" and "invisible extensions"—parts of a program whose semantics change but not in a way that affects existing clients [3]. The combination of backward-compatible and conformable specifications match ideas proposed by Ajmani et al. [1], who studied dynamic updates for distributed systems and proposed mechanisms to maintain continuity for clients of a particular version.

CO-specs can also be used to express the constraints intended by Gupta's *reachability* while side-stepping the problem that reachability can leave behavior under-constrained. For example, for the vsftpd update mentioned above, the programmer can directly write a CO-spec that expresses what should happen to existing client connections, e.g., whether all, some, or none should be preserved. This does not fall into one of the categories above, demonstrating the utility of a full specification language over "one size fits all" notions of update correctness.

Another feature of CO-specs in these categories is that they can be mechanically constructed from CO-specs that are written for a single version. Thus, if a programmer was inclined to verify the correctness of each version of his program

$$
\begin{array}{llll}
\textit{Prog.} & p ::= p, (g, \lambda x.e) \mid \cdot & \textit{Variables} & x, y, z \\
\textit{Exprs.} & e ::= v \mid v_1\ \textit{op}\ v_2 \mid v_1(v_2) \mid\ ? \mid\ !v \mid \mathsf{ref}\ v \mid & \textit{Globals} & f, g \\
 & \quad v_1 := v_2 \mid \mathsf{if}\ v\ e_1\ e_2 \mid \mathsf{update} \mid & \textit{Operators} & \textit{op} \\
 & \quad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{assume}\ v \mid & \textit{Integers} & i, j \\
 & \quad \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \mid \mathsf{assert}\ v \mid & \textit{Addresses} & a \\
 & \quad \mathsf{running}\ p \mid \mathsf{error} & \textit{Heaps} & \sigma \in \textit{Locs} \rightharpoonup \textit{Values} \\
\textit{Values} & v ::= x \mid l \mid i \mid (v_1, v_2) \mid () & \textit{Patch} & \pi ::= (p, e) \\
\textit{Locs.} & l ::= a \mid g & \textit{Labels} & \nu ::= \pi \mid \epsilon
\end{array}
$$

$$
\begin{array}{lll}
\langle p; \sigma; v_1\ \textit{op}\ v_2\rangle & \rightsquigarrow \langle p; \sigma; v'\rangle & v' = [\![\textit{op}]\!](v_1, v_2) \\
\langle p; \sigma; \mathsf{ref}\ v\rangle & \rightsquigarrow \langle p; \sigma[a \mapsto v]; a\rangle & a \notin \textit{dom}(\sigma) \\
\langle p; \sigma; !l\rangle & \rightsquigarrow \langle p; \sigma; v\rangle & \sigma(l) = v \text{ and } l \notin \textit{dom}(p) \\
\langle p; \sigma; a := v\rangle & \rightsquigarrow \langle p; \sigma[a \mapsto v]; v\rangle & a \in \textit{dom}(\sigma) \\
\langle p; \sigma; g := v\rangle & \rightsquigarrow \langle p; \sigma[g \mapsto v]; v\rangle & g \notin \textit{dom}(p) \\
\langle p; \sigma; ?\rangle & \rightsquigarrow \langle p; \sigma; i\rangle & \text{for some } i \\
\langle p; \sigma; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle & \rightsquigarrow \langle p; \sigma; e[v/x]\rangle & \\
\langle p; \sigma; f(v)\rangle & \rightsquigarrow \langle p; \sigma; e[v/x]\rangle & p(f) = \lambda x.e \\
\langle p; \sigma; \mathsf{if}\ 0\ e_1\ e_2\rangle & \rightsquigarrow \langle p; \sigma; e_2\rangle & \\
\langle p; \sigma; \mathsf{if}\ v\ e_1\ e_2\rangle & \rightsquigarrow \langle p; \sigma; e_1\rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{while}\ e_1\ \mathsf{do}\ e_2\rangle & \rightsquigarrow \langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in} & x \notin \textit{fv}(e_1, e_2) \\
 & \qquad \mathsf{if}\ x\ (e_2; \mathsf{while}\ e_1\ \mathsf{do}\ e_2)\ 0\rangle & \\
\langle p; \sigma; \mathsf{update}\rangle & \rightsquigarrow \langle p; \sigma; 0\rangle & \\
\langle p; \sigma; \mathsf{update}\rangle & \overset{\pi}{\rightsquigarrow} \langle p_\pi; \sigma; (e_\pi; 1)\rangle & \pi = (p_\pi, e_\pi) \\
\langle p; \sigma; \mathsf{running}\ p\rangle & \rightsquigarrow \langle p; \sigma; 1\rangle & \\
\langle p; \sigma; \mathsf{running}\ p'\rangle & \rightsquigarrow \langle p; \sigma; 0\rangle & p' \neq p \\
\langle p; \sigma; \mathsf{assume}\ v\rangle & \rightsquigarrow \langle p; \sigma; v\rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{assert}\ v\rangle & \rightsquigarrow \langle p; \sigma; v\rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{assert}\ 0\rangle & \rightsquigarrow \langle p; \sigma; \mathsf{error}\rangle & \\
\langle p; \sigma; \mathsf{let}\ x = \mathsf{error}\ \mathsf{in}\ e\rangle & \rightsquigarrow \langle p; \sigma; \mathsf{error}\rangle &
\end{array}
$$

$$
\frac{\langle p; \sigma; e_1\rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; e_1'\rangle}{\langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2\rangle}
$$

Fig. 2. Syntax and semantics

using CO-specs, the additional work to verify a dynamic update is not much greater. For details, see our technical report [11].

3 Verification via Program Merging

We verify CO-specs by *merging* an existing program version with its update, so that the semantics of the merged program is equivalent to the updating program. This section formalizes a semantics for dynamic updates to single-threaded programs, then defines the merging transformation and proves it correct with respect to the semantics. Many server programs for which dynamic updating is useful are single-threaded [13,19,12]. However, an important next step for this work would be to adapt it to support updates to multi-threaded (and distributed) programs.

3.1 Syntax

The top of Figure 2 defines the syntax of a simple programming language supporting dynamic updates. It is based on the Proteus dynamic update calculus [23], and closely models the semantics of common DSU systems, including Ginseng [19] (which is the foundation of our implementation), Ksplice [16], Jvolve [24], K42 [14], DLpop [13], Dynamic ML [25] and Bracha's DSU system [4].

A *program* p is a mapping from function names g to functions $\lambda x.e$. A function body e is defined by a mostly standard core language with a few extensions for updating. Our language contains a construct update, which indicates a position where a dynamic update may take effect. To support writing specifications, the language includes an expression ?, which represents a random integer, and expressions assume v, assert v, and running p, all of whose semantics are discussed below. Expressions are in administrative normal (A-normal) form [8] to keep the semantics simple—e.g., instead of e_1+e_2, we write let $x = e_1$ in let $y = e_2$ in $x+y$. We write $e_1; e_2$ as shorthand for let $x = e_1$ in e_2, where x is fresh for e_2.

3.2 Semantics

The semantics, given in the latter half of Figure 2, is written as a series of small-step rewriting rules between *configurations* of the form $\langle p; \sigma; e\rangle$, which contain the program p, its current heap σ, and the current expression e being evaluated. A heap is a partial function from locations l to values v, and a location l is either a (dynamically allocated) address a or a (static) global name g. Note that while the language does not include closures, global names g are values, and so the language does support C-style function pointers.[1]

Most of the operational semantics rules are straightforward. We write $e[x/v]$ for the capture-avoiding substitution of x with v in e. We assume that the semantics of primitive operations *op* is defined by some mathematical function $[\![op]\!]$; e.g., $[\![+]\!]$ is the integer addition function. Loops are rewritten to conditionals, where in both cases a non-zero guard is treated as true and zero is treated as false. Addresses a for dynamically allocated memory must be allocated prior to assigning to them, whereas a global variable g is created when it is first assigned to. This semantics allows state transformation functions, described below, to define new global variables that are accessible to an updated program.

The update command identifies a position in the program at which a dynamic update may take place. Semantically, update non-deterministically transitions either to 0, indicating that an update did not occur, or to 1 (eventually), indicating that a dynamic update was available and was applied.[2] In the case where

[1] Variables names x are values so that we can use a simple grammar to enforce A-normal form. The downside is that syntactically well-formed programs could pass around unbound variables and store them in the heap. The ability to express such programs is immaterial to our modeling of DSU, and could be easily ruled out with a simple static type system.

[2] In practice, update would be implemented by having the run-time system check for an update and apply it if one is available [13].

an update occurs, the transition arrow is labeled with the patch π; all other (unadorned) transitions implicitly have label ϵ. A patch π is a pair (p_π, e_π) consisting of the new program code (including unmodified functions) p_π and an expression e_π that transforms the current heap as necessary, e.g., to update an existing data structure or add a new one for compatibility with the new program p_π. In practice, e_π will be a call to a function defined in p_π. The transformer expression e_π is placed in redex position and is evaluated immediately; to avoid capture, non-global variables may not appear free in e_π. Notice that an update that changes function f has no effect on running instances of f since evaluation of their code began prior to the update taking place.

The placement of the update command has a strong influence on the semantics of updates. Placing update pervasively throughout the code essentially models asynchronous updates. Or, as prior work recommends [15,1,19,12], we could insert update selectively, e.g., at the end of each request-handling function or within the request-handling loop, to make an update easier to reason about

The constructs running p, assume v, and assert v allow us to write specifications. The expression running p returns 1 if p is the program currently running and 0 otherwise; i.e., we encode a program version as the program text itself. (In Figure 1(c) the expression is_updated is equivalent to running p where p is the new program version.) The expression assert v returns v if it is non-zero, and error otherwise, which by the rule for let propagates to the top level. Finally, the expression assume v returns v if v is non-zero, and otherwise is stuck.

3.3 Program Merging Transformation

We now present our program merging transformation, which takes an old program configuration $\langle p, \sigma, e\rangle$ and a patch π and yields a single *merged program* configuration, written $\langle p, \sigma, e\rangle \rhd \pi$. We present the transformation formally and then prove that the merged program is equivalent to the original program with the patch applied dynamically. While we focus on merging a program with a single update, the merging strategy can be readily generalized to multiple updates (see our technical report [11] for details).

The definition of $\langle p, \sigma, e\rangle \rhd \pi$ is given in Figure 3(e). It makes use of functions $[\![\cdot]\!]^{\cdot}$ and $\{\!|\cdot|\!\}^{\cdot}$, defined in Figure 3(a)–(d). We present the interesting cases; the remaining cases are translated structurally in the natural way. For simplicity, the transformation assumes the updated program p_π does not delete any functions in p. Deletion of function f can be modeled by a new version of f with the same signature as the original and the body assert(0).

The merging transformation renames each new-version function from g to g', and changes all new-version code to call g' instead of g (the first rewrite rules in Figure 3(b) and (d), respectively). For each old-version function g, it generates a new function g_{ptr} whose body conditionally calls the old or new version of g, depending on whether an update has occurred (Figure 3(a)). The transformation introduces a global variable *uflag* (Figure 3(e)) and a function *isupd* to keep track of whether the update has taken place (bottom of Figure 3(a)). All calls to g in the old version are rewritten to call g_{ptr} instead (top of Figure 3(c)).

$$
\begin{aligned}
&[\![p', (g, \lambda y.e)]\!]^{p,\pi} \triangleq \\
&\qquad [\![p']\!]^{p,\pi},\ (g, \lambda y.[\![e]\!]^{p,\pi}), \\
&\qquad (g_{ptr}, \lambda y.\mathsf{let}\ z = isupd()\ \mathsf{in\ if}\ z\ g'(y)\ g(y)) \\
&[\![\cdot]\!]^{p,\pi} \triangleq (\cdot, (isupd, \lambda y.\mathsf{let}\ z = !uflag\ \mathsf{in}\ z > 0))
\end{aligned}
$$

(a) Old version programs

$$
\begin{aligned}
&\{\!|p', (g, \lambda y.e)|\!\}^{p} \triangleq \\
&\qquad \{\!|p'|\!\}^{p}, (g', \lambda y.\{\!|e|\!\}^{p}) \\
&\{\!|\cdot|\!\}^{p} \triangleq \cdot
\end{aligned}
$$

(b) New version programs

$$
\begin{aligned}
&[\![g]\!]^{p,\pi} \triangleq \\
&\qquad \begin{cases} g_{ptr} & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
&[\![\mathsf{running}\ p'']\!]^{p,(p_\pi,e_\pi)} \triangleq \\
&\qquad \begin{cases} \mathsf{let}\ z = isupd()\ \mathsf{in}\ z = 0 & \text{if } p = p'' \\ isupd() & \text{if } p_\pi = p'' \\ 0 & \text{otherwise} \end{cases} \\
&[\![\mathsf{update}]\!]^{p,(p_\pi,e_\pi)} \triangleq \\
&\qquad \mathsf{let}\ z = isupd()\ \mathsf{in} \\
&\qquad \mathsf{if}\ z\ 0\ (uflag := ?; \\
&\qquad\qquad \mathsf{let}\ z = isupd()\ \mathsf{in\ if}\ z\ (\{\!|e|\!\}^{p_\pi}; 1)\ 0)
\end{aligned}
$$

(c) Old version expressions

$$
\begin{aligned}
&\{\!|g|\!\}^{p} \triangleq \\
&\qquad \begin{cases} g' & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
&\{\!|\mathsf{running}\ p''|\!\}^{p} \triangleq \\
&\qquad \begin{cases} 1 & \text{if } p = p'' \\ 0 & \text{otherwise} \end{cases} \\
&\{\!|\mathsf{update}|\!\}^{p} \triangleq 0
\end{aligned}
$$

(d) New version expressions

$$
\begin{aligned}
&\langle p; \sigma; e\rangle \ \rhd\ \pi \triangleq \langle \overline{p}, \overline{\sigma}[uflag \mapsto i], \overline{e}\rangle \\
&\qquad \text{where } (p_\pi, e_\pi) = \pi \qquad \overline{p} = \{\!|p_\pi|\!\}^{p_\pi}, [\![p]\!]^{p,\pi} \qquad \overline{e} = [\![e]\!]^{p,\pi} \\
&\qquad\qquad i \le 0 \qquad \overline{\sigma} = \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\}
\end{aligned}
$$

(e) Merging a configuration and a patch

Fig. 3. Merging transformation (partial)

The transformation rewrites occurrences of update in old-version code into expressions that check whether *uflag* is positive (bottom of Figure 3(c)). If it is, then the update has already taken place, so there is nothing to do. Otherwise, the transformation sets *uflag* to ?, which simulates a non-deterministic choice of whether to apply the update. If *uflag* now has a positive value, the update path was chosen, so the transformation executes the developer-provided state transformation e, which must also be transformed according to $\{\!|\cdot|\!\}^{\cdot}$ to properly reference functions in the new program. While this transformation results in multiple occurrences of the expression e, in practice e is a call to a state transformation function defined in the new version and so does not significantly increase code size.

Version tests running p are translated into calls to *isupd* in the old version, and to appropriate constants in the new code (since we know the update has occurred if new code is running).

3.4 Equivalence

We can now prove that an update to an old-program configuration is correct if and only if the result of merging that configuration and the update is correct.

This result lets us use stock verification tools to check properties of dynamic updates using the merged program, which simulates updating, instead of having to develop new tools or extend existing ones.

We say that a program and a sequence of updates are *correct* if evaluation never reaches error (i.e., if there are no assertion failures). More formally:

Definition 1 (Correctness). *A configuration* $\langle p; \sigma; e\rangle$ *and an update* π *are* correct*, written* $\models \langle p; \sigma; e\rangle, \pi$*, if and only if for all* p', σ', e' *it is the case that* $\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow}^{*} \langle p'; \sigma'; e'\rangle$ *implies* e' *is not* error*.*

The expression e at startup could be a call to an entry-point function (i.e., main). A correct program need not apply π, though no other update may occur. When no update is permitted we write $\models \langle p; \sigma; e\rangle$.

Theorem 1 (Equivalence). *For all* p, σ, e, π *such that* $dom(p_\pi) \supseteq dom(p)$ *we have that* $\models \langle p; \sigma; e\rangle, \pi$ if and only if $\models (\langle p, \sigma, e\rangle \ \triangleright \ \pi)$.

The proof is by bisimulation and is detailed in our technical report [11].

Observe that type errors result in stuck programs, e.g., !1 does not reduce, while the above theorem speaks only about reductions to error. We have chosen not to consider type safety in the formal system to keep things simple; adding types, we could appeal to standard techniques [23,24,25,7]. Our implementation catches type errors that could arise due to a dynamic update by transforming them into assertion violations. In particular, we rename functions and global variables whose type has changed prior to merging, essentially modeling the change as a deletion of one variable and the addition of another. Deleted functions are modeled as mentioned above, and deleted global variables are essentially assigned the error expression. Thus, any old code that accesses a stale definition post-update (including one with a changed type) fails with an assertion violation.

4 Experiments

To evaluate our approach, we have implemented the merging transformation for C programs, with the additional work to handle C being largely routine. We merged several programs and dynamic updates and then checked the merged programs against a range of CO-specs. We analyzed the merged programs using two different tools: the symbolic executor Otter, developed by Ma et al. [22], and the verification tool Thor, developed by Magill et al. [18]. The tools represent a tradeoff: Otter is easier to use and more scalable but provides incomplete assurance, while Thor can guarantee correctness but is less scalable and requires more manual effort. Overall, both tools proved useful. Otter successfully checked all the COs-specs we tried, generally in less than one minute. Thor was able to fully verify several updates, though running times were longer. Both tools found bugs in updates, including mistakes we introduced inadvertently. On average, verification of merged code took four times longer than verification of a single version. Since our approach is independent of the verification tool used, its performance and effectiveness will improve as advances are made in verification technology.

Program – *change*	**Thor time (s)**			**Otter time (s)**		
CO-specs	**old**	**new**	**mrg**	**old**	**new**	**mrg**
Multiset – *disallow duplicates* (correct)						
mem-mem^b	90.11	121.27	1003.22	6.29	9.72	49.37
add-mem^b	64.17	89.71	537.01	3.26	10.48	50.84
add-add-del-set^g			–			4.04
Multiset – *disallow duplicates* (broken)						
mem-mem^b	25.33	57.78	133.68	6.28	9.77	42.5
add-mem^b	15.68	33.50	80.07	3.25	9.94	33.53
add-add-del-set-fails^g			122.71			5.49
Key-value store – *bug fix*						
put-get^b	27.01	26.13	41.62	3.28	2.54	18.42
new-def-shadows^g			–			4.19
new-def-shadows-bc-fails^b	38.97	41.52	117.56	3.88	2.06	19.03
Key-value store – *added namespaces*						
new-def-shadows-post^p		–	–		1.02	2.99
put-get^p		–	–		18.32	228.69
new-def-shadows-conf^c	–	–	–	1.19	1.93	7.53
put-get-conf^c	–	–	–	4.23	7.09	61.41
Key-value store – *optimization* (broken)						
put-get-back^b	42.133	–	–	2.08	11.01	56.44
new-def-shadows-back^b	15.344	–	–	2.14	11.33	56.03
Key-value store – *optimization* (correct)						
put-get-back^b	41.87	–	–	2.07	10.87	69.31
new-def-shadows-back^b	15.72	–	–	2.14	10.96	68.95

b – backward compatible p – post update c – conformable g – general
A dash indicates that the example could not be verified.

Fig. 4. Synthetic examples

4.1 Programs

We ran Otter and Thor on updates to three target programs. The first two are small, synthetic examples: a multiset server, which maintains a multiset of integer values, and a key-value store. For each program, we also developed a number of updates inspired by common program changes such as memory and performance optimizations and semantic changes observed in real-world systems such as Cassandra [5]. The third program we considered is Redis [21], a widely used open-source key-value server. At roughly 12k lines of C code, Redis is significantly larger that our synthetic examples, and is currently not tractable for Thor. We developed six dynamic patches for Redis that update between each pair of consecutive versions from 1.3.6 through 1.3.12, and we also wrote a set of CO-specs that describe basic correctness properties of the updates.

As we mention in Section 2, we join each CO-spec with the server code and have the main function invoke the CO-spec after it initializes server data structures. The new-version source code includes the state transformation code, which is identified by a distinguished function name recognized by the merger.

Synthetic Examples. Figure 4 lists the synthetic benchmarks we constructed for our multiset and key-value store programs. Each grouping of rows shows a dynamic update and a list of CO-specs we wrote for that update. The multiset program has routines to add and delete elements and to test membership. The updates both change to a set semantics, where duplicate elements are disallowed. The first (correct) state transformer removes all duplicates from a linked list that maintains the current multiset. The second update has a broken state transformer that fails to remove duplicates.

The key-value store program also implements its store with a linked list. The updates are inspired by code changes we have seen in practice and include a bug fix (bindings could not be overwritten), a feature addition (adding namespaces), and an optimization (removing overwritten bindings), where for this last update the state transformer was broken at first.

The properties span all the categories of CO-specs that we outlined in Section 2. Backward compatible specs, such as add-mem, check core functionality that does not change between versions (add actually adds elements, delete removes elements, etc.). Post-update and general CO-specs are used to check that functionality *does* change, but only in expected ways. For example, new-def-shadows in the *bug-fix* update checks that, following the update, new key-value bindings properly overwrite old bindings (which was not true in the old version).

We wrote specifications to be as general as possible. For example, add-mem, on the second line of the table in Figure 4, checks that after an element is added, it is reported as present after an arbitrary sequence of function calls that does not include delete(). The code for our synthetic examples and their associated CO-specs is available on-line.[3]

Redis. Figure 5 lists the updates and CO-specs for Redis. Four of the six updates required writing state transformers, often just to initialize added fields but sometimes to perform more complex transformation, e.g., the update to 1.3.9 required some reorganization of data structures storing the main database.

We found that across these updates, there were four different kinds of behavioral changes, each of which suggested a certain strategy for developing CO-specs; we employed CO-specs in each of the classes described in Section 2:

– *Unmodified behavior*: We adapted two CO-specs from our synthetic key-value store example (Figure 4), *put-get* and *new-def-shadows*, to check correct behavior of Redis' SET and GET operations over string values. As these CO-specs concern behavior that all versions of Redis should exhibit, we applied them as backward compatible CO-specs.
– *New operations*: The HASHINCRBY operation, which adds to the numeric value stored for a hash key, first appeared in version 1.3.8. We check the operation's correctness using a post-update CO-spec, *hashincrby*. The HASHINCRBY operation is supported by all later versions, and so we also developed a backward compatible *hashincrby* CO-spec for subsequent updates.

[3] `http://www.cs.umd.edu/projects/PL/dsu/data/dsumerge-examples.tar.gz`

		Otter time (s)		
	Specification	old	new	mrg
→1.3.7	put-get^b	9.76	9.52	24.99
	new-def-shadows^b	2.19	2.19	3.97
	empty-set-exists^b	9.95	9.92	29.15
→1.3.8*	put-get^b	9.20	9.58	28.53
	new-def-shadows^b	2.17	2.27	4.14
	hashincrby^p		3.02	14.81
	empty-set-notexists^g			27.58
→1.3.9*	put-get^b	9.14	9.31	48.08
	new-def-shadows^b	2.27	2.66	5.46
	hashincrby^b	14.23	14.83	77.14
	empty-set-notexists^b	10.56	11.13	62.88

b – backward compatible p – post update
c – conformable g – general $*$ – xform

		Otter time (s)		
	Specification	old	new	mrg
→1.3.10	put-get^b	9.22	10.05	27.37
	new-def-shadows^b	2.70	2.69	4.79
	hashincrby^b	14.86	15.26	46.74
	empty-set-notexists^b	11.14	11.36	35.01
→1.3.11*	put-get^b	9.85	10.04	50.73
	new-def-shadows^b	2.69	2.77	6.30
	hashincrby^b	15.19	15.51	77.80
	empty-set-notexists^b	11.33	11.57	72.40
→1.3.12*	put-get^b	10.32	9.72	49.23
	new-def-shadows^b	2.85	2.92	6.27
	hashincrby^b	15.20	14.79	77.27
	empty-set-notexists^b	11.58	11.67	72.16
	zinter^c	60.30	59.73	294.05

Fig. 5. Otter checking times for Redis

– *Modified semantics*: Before Redis version 1.3.8, a set whose last element was removed would remain in the database. We use the backward compatible CO-spec *empty-set-exists* to check this property against the patch to 1.3.7. Then for the patch to 1.3.8, which causes the server to remove a set when it becomes empty, we use a general CO-spec *empty-set-notexists* to ensure that sets are removed if they become empty after the update. Subsequent versions preserve this behavior, which we specify using a backward compatible CO-spec.
– *Conformable changes*: Redis's ZINTER operation, which computes the intersection of two sorted sets, was renamed to ZINTERSTORE in version 1.3.12. We use a conformable CO-spec, *zinter*, to specify correct behavior regardless of when an update occurs.

To make symbolic execution tractable for Redis, we had to bound the nondeterminism in our CO-specs, e.g., by limiting "arbitrary behavior" to a single operation, non-deterministically chosen from a subset of commands that relate to the specified property (rather than from the full set of Redis operations).

4.2 Effectiveness

In most cases, checking CO-specs validated the correctness of our dynamic patches. In some cases the checking found bugs. For example, in the state transformer for the multiset-to-set update, we inadvertently introduced a possible null pointer dereference when freeing duplicates. Verification with Thor discovered this problem. For Redis, we experimented with omitting state transformation code or using code with a simple mistake in it. In all cases, checking our specifications with Otter uncovered the mistakes.

Figures 4 and 5 show the running times for each of the update/CO-spec/tool combinations, listed under the **mrg** heading. As a baseline, we also list the running times for the backward-compatible specifications on both individual program versions, and for post-update specifications on the new version—this lets us compare the relative slowdown incurred by reasoning about updates.

Otter. We performed experiments with Otter on a machine with a dual-core Pentium-D 3.6GHz processor and 2GB of memory. The running times range from seconds to a few minutes, depending on the complexity of the specification and the program. For example, the CO-specs for the multiset-to-set example were expensive to symbolically execute because each set insertion checks for duplicates, which induces many branches when symbolic values are involved.

We also see that, across the synthetic examples and Redis, it takes four times longer to analyze merged programs versus individual versions on average, and 6.4 times longer in the worst case. We investigated the source of the slowdown, and found it was due to the extra time required to model update points and state transformers, which is fundamental to verifying updating programs, rather than an artifact of our merging strategy. In particular, Otter runs on the merged versions, so it must explore additional program paths to model each possible update timing; on average, CO-specs reached 3.7 update points during execution and, loosely speaking, each update point could induce another full exploration through the set of non-updating program paths. State transformation is also executed following updates, so the expense of symbolically executing the transformer is multiplied by the number of times an update point is reached. Nevertheless, despite this slowdown, total checking time was rarely an impediment to checking useful properties.

Thor. We ran Thor on a 2.8GHz Intel Core 2 Duo with 4GB of memory. The average slowdown was 3.9 times, and ranged from 1.5 times to 8.3 times. Much of the slowdown derived from per-update-point analysis of the state transformation function; tools that compute procedure summaries or otherwise support modular verification would likely do better. Thor could not verify all our examples, owing to complex state transformation code and CO-specs that specify very precise properties. For example, for the multiset-to-set example, Thor was able to prove that the state transformer preserves list membership (used to verify *mem-mem*), but not that it leaves at most one copy of any element in the list (needed for *add-add-del-set*).

The CO-specs we considered lie at the boundary of what is possible for current verification technology. To verify all our examples requires a robust treatment of pointer manipulation, integer arithmetic, and reasoning about collections. We are not aware of any tools that currently offer such a combination. However, we hope that the demonstrated utility of such specifications will help inspire further research in this area.

5 Related Work

This paper presents the first approach for automatically verifying the correctness of dynamic software updates. As mentioned in the introduction, prior automated analyses focus on safety properties like type safety [23], rather than correctness.

As described in Section [2], our notion of client-oriented specifications captures and extends prior notions of update correctness.

Our verification methodology generalizes our prior work [10,12] on systematically *testing* dynamic software updates. Given tests that pass for both the old and new versions, the tool tests every possible updating execution. This approach only supported backward-compatible properties and does not extend to general properties (e.g., with non-deterministically chosen operations or values).

The merging transformation proposed in this paper was inspired by KISS [20], a tool that transforms multi-threaded programs into single-threaded programs that fix the timing of context switches. This allows them to be analyzed by non–thread-aware tools, just as our merging transformation makes dynamic patches palatable to analysis tools that are not DSU-aware.

An alternative technique for verifying dynamic updates, explored by Charlton et al. [6], uses a Hoare logic to prove that programs and updates satisfy their specifications, expressed as pre/post-conditions. We find CO-specs preferable to pre/post-conditions because they require less manual effort to verify, and because they naturally express rich properties that span multiple server commands.

6 Summary

We have presented the first system for automatically verifying dynamic-software-update (DSU) correctness. We introduced *client-oriented specifications* as a way to specify update correctness and identified three common, easy-to-construct classes of DSU CO-specs. To permit verification using non-DSU-aware tools, we developed a technique where the old and new versions are *merged* into a single program and proved that it correctly models dynamic updates. We implemented merging for C and found that it enabled the analysis tool, Thor, to fully verify several CO-specs for small updates, and the symbolic executor, Otter, to check and find errors in dynamic patches to Redis, a widely-used server program.

Acknowledgements. We thank Elnatan Reisner, Matthew Parkinson, Nishant Sinha, and the anonymous reviewers for helpful comments on drafts of this paper. This research was supported by the partnership between UMIACS and the Laboratory for Telecommunications Sciences, by ONR grant N00014-09-1-0652, and NSF grants CCF-0910530, CCF-0915978 and CNS-1111698. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR or NSF.

References

1. Ajmani, S., Liskov, B., Shrira, L.: Modular Software Upgrades for Distributed Systems. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 452–476. Springer, Heidelberg (2006)
2. Armstrong, J., Virding, R., Wikstrom, C., Williams, M.: Concurrent programming in ERLANG, 2nd edn. Prentice Hall International Ltd. (1996)

3. Bloom, T., Day, M.: Reconfiguration and module replacement in Argus: theory and practice. Software Engineering Journal 8(2), 102–108 (1993)
4. Bracha, G.: Objects as software services (August 2006), http://bracha.org/objectsAsSoftwareServices.pdf
5. Cassandra API overview, http://wiki.apache.org/cassandra/API
6. Charlton, N., Horsfall, B., Reus, B.: Formal reasoning about runtime code update. In: HOTSWUP (2011)
7. Duggan, D.: Type-based hot swapping of running modules. In: ICFP (2001)
8. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: PLDI (1993)
9. Gupta, D., Jalote, P., Barua, G.: A formal framework for on-line software version change. IEEE TSE 22(2) (1996)
10. Hayden, C.M., Hardisty, E.A., Hicks, M., Foster, J.S.: Efficient Systematic Testing for Dynamically Updatable Software. In: HOTSWUP (2009)
11. Hayden, C.M., Magill, S., Hicks, M., Foster, N., Foster, J.S.: Specifying and verifying the correctness of dynamic software updates (extended version). Technical Report CS-TR-4997, Dept. of Computer Science, University of Maryland (2011)
12. Hayden, C.M., Smith, E.K., Hardisty, E.A., Hicks, M., Foster, J.S.: Evaluating dynamic software update safety using systematic testing (March 2011)
13. Hicks, M., Nettles, S.: Dynamic software updating. ACM TOPLAS 27(6) (2005)
14. The K42 Project, http://www.research.ibm.com/K42/
15. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE TSE 16(11) (1990)
16. Never reboot Linux for Linux security updates : Ksplice, http://www.ksplice.com
17. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: THOR: A Tool for Reasoning about Shape and Arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
18. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
19. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: PLDI (2006)
20. Qadeer, S., Wu, D.: KISS: Leep it simple and sequential. In: PLDI (2004)
21. The Redis project, http://code.google.com/p/redis/
22. Reisner, E., Song, C., Ma, K.-K., Foster, J.S., Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In: ICSE (2010)
23. Stoyle, G., Hicks, M., Bierman, G., Sewell, P., Neamtiu, I.: Mutatis Mutandis: Safe and flexible dynamic software updating. ACM TOPLAS 29(4) (2007)
24. Subramanian, S., Hicks, M., McKinley, K.S.: Dynamic software updates for Java: A VM-centric approach. In: PLDI (2009)
25. Walton, C.: Abstract Machines for Dynamic Computation. PhD thesis, University of Edinburgh, ECS-LFCS-01-425 (2001)

Symbolic Execution Enhanced System Testing

Misty Davies[1], Corina S. Păsăreanu[2], and Vishwanath Raman[2]

[1] NASA Ames Research Center, Moffett Field CA 94035, USA
misty.d.davies@nasa.gov
[2] Carnegie Mellon University, Moffett Field, CA 94035, USA
corina.s.pasareanu@nasa.gov, vishwa.raman@west.cmu.edu

Abstract. We describe a testing technique that uses information computed by symbolic execution of a program unit to guide the generation of inputs to the system containing the unit, in such a way that the unit's, and hence the system's, coverage is increased. The symbolic execution computes unit constraints at run-time, along program paths obtained by system simulations. We use machine learning techniques –treatment learning and function fitting– to approximate the system input constraints that will lead to the satisfaction of the unit constraints. Execution of system input predictions either uncovers new code regions in the unit under analysis or provides information that can be used to improve the approximation. We have implemented the technique and we have demonstrated its effectiveness on several examples, including one from the aerospace domain.

1 Introduction

Modern software, and in particular flight control software like that written at NASA, needs to be highly reliable and hence thoroughly tested. NASA software is typically tested using system level Monte Carlo or combinatorial simulations. Such system level "black-box" simulations have the advantage that they are (a) easy to set up, since the user only needs to specify the ranges for the system level inputs, and (b) can be used to test software systems that contain COTS ("Commercial-Off-The-Shelf"), binary or even hardware components that are impervious to "white-box" methods. However, system level simulations provide few guarantees in terms of testing coverage. Furthermore, they may be quite expensive. For example, a run using NASA's ANTARES simulator [1] may take hours to complete.

Recently, a new set of techniques [2,3,4] based on symbolic execution [5] have emerged for generating test cases that achieve high code coverage. Symbolic execution and its variant, concolic execution, are white-box as they collect constraints based on the *internal* code structure. The collected constraints are solved systematically to obtain inputs that exercise all the paths through the code (up to some user specified bound). Such white-box techniques are not effective in the presence of COTS or binary components; e.g., in such cases, concolic execution may lead to divergence [4]. For this reason, and due to the large number of paths to explore and complex constraints to be solved, white-box symbolic execution is used most effectively for testing individual software units, but not the whole system. On the other hand, when analyzing a unit in isolation, it is often the case that the unit's inputs need to be constrained by the system calling context, in order to obtain realistic test cases. Encoding input constraints requires significant manual effort by developers [2].

The goal of our work is to find system level test cases that increase the coverage of a unit of interest by exploiting a synergy between black-box system simulation and white-box unit symbolic execution. We propose an iterative procedure that uses the information computed by a symbolic execution of a unit to *guide*, via machine learning techniques, the generation of new system level inputs that increase the coverage of the unit, and hence of the system containing the unit. Thus, our approach improves on system level testing by increasing the obtained coverage with a reduced number of tests, and hence with a reduced cost. It also enables a modular unit level analysis under *realistic* contexts, since symbolic execution is performed along the program paths obtained via simulation.

Specifically, we use data mining techniques (i.e. treatment learning [6]) to obtain an approximation of the system level input constraints that influence the satisfaction of the unit level constraints computed by the symbolic execution of the unit. Function fitting is performed to incrementally approximate the behavior of the unit's calling context. Finally, the unit level constraints are solved with off-the shelf constraint solvers and, together with the approximations, are used to guide the generation of new system level inputs towards executing uncovered code regions in the unit under analysis. We have implemented the techniques in the context of the analysis of C programs. We report here on the application of our approach to several illustrative examples, including one from the aerospace domain.

Related Work. The work related to automated testing is vast and we only highlight here the work that is most related to our approach. We have already discussed related symbolic and concolic execution approaches [7,4,3,8]. The work on carving differential unit tests from system tests [9] extracts the components that influence the execution of a unit and reassembles them so that the unit can be exercised as it was by the system test. Differential unit tests are used to detect differences between multiple unit implementations; they can not be used to guide the system level inputs to increase coverage.

In previous work [2] we described a symbolic execution framework that used system level simulations to improve the precision of symbolic execution at the unit level. This was achieved in two ways: first, the framework allows symbolic execution to be started at any point in the program; thus, the concrete execution of the system can be effectively used to set up the environment for the symbolic execution of a unit in the system. However, that work could not be used for *guiding* the generation of new system level inputs to increase the coverage of the unit—which is our contribution here. Furthermore, we described in [2] how to use the data collected during system level runs to mine constraints on the unit level inputs (using treatment learning or Daikon, for example). While this approach would allow more focused unit level testing, it suffers from the drawback that the mined constraints can be unrealistically restrictive, and thus prevent us to achieve coverage of corner cases in the unit.

2 Background

A Program Model. A program is a tuple $P = (I, A, C)$, where I is a set of input parameters, A is a set of assignment statements and C is a set of conditional statements. We assume that the elements of I are of *basic types*, defined to be a type from the set $\{int, short, unsigned\ int, char, float, double, enum\}$, with each element $a \in I$ taking values from a domain D_a based on its type; all assignment and conditional statements refer to elements in I. The set of all executions of the program P is $R(P) \subseteq \{(A \cup C)^*\}$ – a set of finite sequences of assignments and conditional statements visited over all possible values of the parameters in I. An assignment over the parameters in I, called a *valuation*, is denoted by $\boldsymbol{I}$ and associates every element $a \in I$ to a value in D_a. Given a valuation $\boldsymbol{I}$, we assume that all executions of the program visit exactly the same finite sequence of assignments and conditional statements; the programs are deterministic.

Concolic Execution. Concolic execution [4,10] is a technique that combines concrete and symbolic program execution to increase path coverage. Symbolic path constraints (PCs) are collected along concrete program runs; the PCs are conjunctions of Boolean expressions, each expression representing the condition on the inputs to follow that particular path. The conditions in the PCs are systematically negated to generate new PCs that are solved with off-the-shelf solvers. The obtained solutions are used as new program inputs to run the program along different paths. The process terminates when all the paths have been resolved or a user-specified bound has been reached; paths are either covered, unsatisfiable or unsolvable due to limitations in the chosen solvers.

Treatment Learning. Treatment learning [6,11] is a machine learning technique that finds the minimal difference between two sets. In our work, we use treatment learning to determine a *small number* of controllable inputs and ranges (a *treatment*) that are most likely to lead to some output.

TAR3 is a treatment learner that finds association rules involving both continuous and discrete variables quickly [11]. Given a data set and a partition of that set into a set of desired data points and a set of all remaining points, TAR3 looks for rules (subsets of input parameters and their ranges) that maximize the likelihood of seeing points in the desired set. We note that one can use other association rule learners [12,13,14] to potentially find more accurate rules; however this would come with greater complexity and time costs [15,16].

Function Fitting. Function fitting finds a predictive relationship between associated outputs and inputs (usually one output variable and a small number of inputs). We use discrete least-squares function fitting [17,18] to approximate a relationship between the unit inputs and the associated system inputs; the technique is less sensitive to outliers than many competing techniques [19]. Assume $y(x)$ is a complex, non-linear function; its approximation can be given by a polynomial $p(x)$ with coefficients c_i, for $i \in \{1, 2, 3, \ldots\}$. A least-squares solution finds the constant values c_i that minimize the total Euclidean distance (the *residual*) between $p(x)$ and $y(x)$ at the given measurements x. If the relationships we are trying to approximate are Lipschitz continuous (or *smooth*), we can find a polynomial approximation that is arbitrarily close to our desired function by the Weierstrass Approximation Theorem [20]. A function that is not smooth

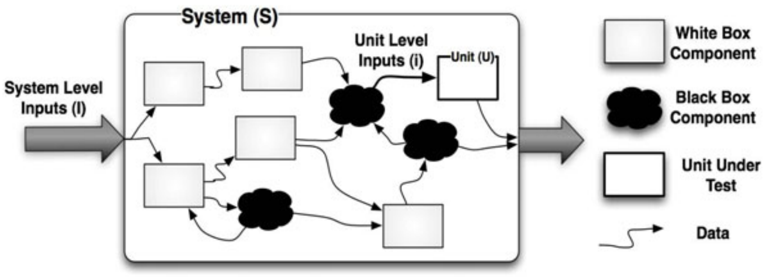


Fig. 1. A system S with inputs I and an embedded unit U with inputs i

along its entire domain may be *locally smooth*, or smooth along some subinterval of the domain. A polynomial constructed on this subset is known as a *piecewise-polynomial approximation*. Shrinking each subinterval allows for arbitrarily close approximations with low-order polynomials [21]. We use the term *Threshold* to represent the minimum number of data points that we need for function fitting.

3 Approach

We illustrate the proposed approach using Figure 1, which shows a System Under Test (S) that may have both white-box and black-box components. A white-box unit is a code fragment that lends itself to concolic testing. S is a system with input parameters I containing a white-box unit $U = (i, A, C)$ with unit level parameters i. The goal is to generate system level inputs I that increase the coverage of unit U.

Let $c \in C$ denote some conditional statement in U that was not covered during system level testing. Let $Cons(c)$ denote the unit level constraint, over parameters in i, associated with statement c; this constraint is obtained by the concolic execution of U. As an example, if $i = \{v, w\}$, a constraint could be $(v > w)$. We note that the concolic execution of U (in isolation) excludes the system that instantiates U; while this is useful for discovering new constraints for the uncovered paths, it may also generate an over-approximation of the actual paths that can be covered during system level testing. By the same token, paths that are unreachable in U remain unreachable in S; a path unreachable in the most liberal environment for U remains unreachable in S. If $Cons(c)$ is satisfiable, then a satisfying valuation $\boldsymbol{i}$ will enable us to cover statement c at the unit level, but as mentioned, that statement may still be unreachable at the system level. Our goal is to try to generate assignments over the system level parameters I that can cover c (and other statements in the unit) during *system level testing*.

We note that the calling context for the unit can be represented by some function f such that $i = f(I)$. To discover the new valuations for I, we monitor the values of I and i during simulations and use machine learning techniques to approximate f, based on the monitored values. Once we have an approximation p of f, we use it to solve $i = p(I) \wedge Cons(c)$; the solutions for I are the likely candidates to the system level inputs that lead to the satisfaction of $Cons(c)$. These valuations are used to start new simulation runs, which lead to either covering c or to obtaining a more accurate

Program 1 . Prototype Linear Example

```
int g1 = 1, g2 = 2;
int System(int I1, int I2) {
  if (I1 > 0) g1 = I2; else g1 = -I2;
  g2 = I1 + 3;
  Unit(I2, I1);
}
int Unit(int i1, int i2) {
  if(i1 > 0) {
    i2 = g2;
    if(i2 > 0) return 0; else return 1;
  } else {
    i2 = g1 + 3;
    if(i2 > 0) return 2; else return 3;
  }
}
```

approximation of f. The process is repeated until either the desired coverage is obtained or a user-specified bound has been reached. We note here that if the function relating I and i is invertible, one can learn an approximation of the form $I = p(i)$ and use the solutions of $Cons(c)$ to directly obtain the valuations of I. To simplify the presentation, we will assume for the rest of the paper that we have such invertible functions. We describe our approach in detail in the next section.

4 Testing Algorithms

As a running example, consider the linear code in listing Program 1. Integers $I1$ and $I2$ are the system inputs, while $i1$ and $i2$ are the unit inputs. The two integer global variables $g1$ and $g2$ are treated as inputs to both System and Unit. The unit inputs are therefore $i1$, $i2$, $g1$ and $g2$.

Constraints Trees. We assume concolic execution achieves full path coverage over Unit. The set of path constraints over all executions of Unit are stored in a *constraints tree* T. The constraints tree reflects the set of all paths that were taken by all executions of a program unit (assume that the unit has no infinite loops).

```
[Parameters]
i1
g2
g1
[Tree]
  (i1 > 0) (C)
    (g2 > 0) (C)
    (g2 <= 0) (S)
  (i1 <= 0) (C)
    ((g1 + 3) > 0) (C)
    ((g1 + 3) <= 0) (S)
```

Fig. 2. The constraints tree after some rounds of initial testing

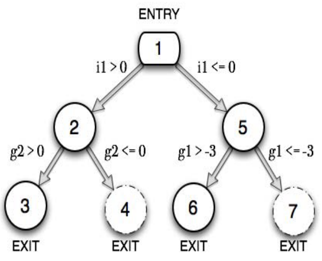


Fig. 3. A graphical representation of Figure 2. Covered nodes are solid circles; those not covered are dotted.

Figure 2 shows T for Unit after some initial testing. Lines 2-4 list the inputs that are constrained. Lines 6-11 contain a textual representation of the tree. The number of leaves is equal to the number of path constraints in T; each path constraint is a conjunction of the terms encountered along the parent hierarchy starting at each leaf. Therefore, given the tree in Fig 2, the set of constraints are: $(i1 > 0) \wedge (g2 > 0)$, $(i1 \leq 0) \wedge (g2 \leq 0)$, $(i1 \leq 0) \wedge ((g1+3) > 0)$ and $(i1 \leq 0) \wedge ((g1+3) \leq 0)$. Of these constraints, $(i1 > 0) \wedge (g2 > 0)$ and $(i1 \leq 0) \wedge ((g1 + 3) > 0)$ were covered during our initial testing, denoted by the letter "C" within parentheses. The other constraints are satisfiable at the Unit level but not covered during system level testing, denoted by the letter "S".

Observations. Consider again a system S, with system inputs I, and a unit U within S, with unit inputs i. We let $d = |I|$. We assume the unit can be fully analyzed using concolic execution. Let T be a constraints tree extracted by monitoring U during system level testing. Consider nodes in T that are satisfiable at the unit level but not covered by system level testing. We attempt to cover such nodes using a combination of concolic execution, treatment learning and function fitting. For a node n in T we take $Cons(n)$ as the unit constraint that leads to n and that when satisfiable will cover n. To present our coverage algorithm, we first make the following observations.

Consider a path $\sigma = n_1, n_2, \ldots, n_k$ in T such that all nodes n_i for $1 \leq i \leq k$ are covered by system testing. There exist vectors at the system and unit level that witness covering each node n_i in σ; for a set of system vectors V_i that witness covering n_i in σ, there exist corresponding witnesses v_i of unit vectors. We then have the following properties of these witnesses:

Observation 1 (Monotonicity of Witnesses). For a constraints tree T and a path $\sigma = n_1, n_2, \ldots, n_k$ of nodes in T, such that $n_1, n_2, \ldots, n_k$ are covered with witness sets $V_1, V_2, \ldots, V_k$ at the system level and corresponding sets $v_1, v_2, \ldots, v_k$ at the unit level, we have, $V_1 \supseteq V_2 \supseteq \ldots \supseteq V_k$ and $v_1 \supseteq v_2 \supseteq \ldots \supseteq v_k$.

Monotonicity of Witnesses follows easily by noting that $Cons(n_k) \Rightarrow Cons(n_{k-1}) \Rightarrow \ldots \Rightarrow Cons(n_1)$ for the constraints of nodes in σ.

Observation 2 (Sufficiency of Witnesses). For a constraints tree T and a path $\sigma = n_1, n_2, \ldots, n_k$ of nodes in T, such that $n_1, n_2, \ldots, n_k$ are covered with witness sets $V_1, V_2, \ldots, V_k$ at the system level and corresponding sets $v_1, v_2, \ldots, v_k$ at the unit level, let $|V_j| \geq Threshold$ such that for all $i \in [1, k]$ with $|V_i| \geq Threshold$, we have $|V_j| \leq |V_i|$. If the relation between V_j and v_j is smooth for function fitting, then for all $i \geq j$, the relation between V_i and v_i is also smooth for function fitting.

Consider T and a $\sigma = n_1, n_2, \ldots, n_k$ in T such that all nodes that precede n_k are covered during system testing, but node n_k is not covered. Since concolic execution fails at the system level, we have that $Cons(n_k)$ is the finest symbolic path constraint, such that when $Cons(n_k)$ is satisfiable, the assignment that satisfies $Cons(n_k)$ covers n_k at the unit level. We take $Term(n_k)$ as the term corresponding to n_k and $Parent(n_k)$ as the parent of n_k in σ. Given a constraint C, let $Vars(C)$ be the set of parameters that appear in the terms of constraint C. The path constraint $Cons(n_k)$ is then $Term(n_1) \wedge Term(n_2) \wedge \ldots \wedge Term(n_k)$. We would like to learn the system level behavior as a function f, such that $I = f(Vars(Cons(n_k))$, via function fitting. If $Cons(n_k)$

is satisfiable, we can use f to find a system level vector that covers n_k using the satisfying assignment over $\mathit{Vars}(\mathit{Cons}(n_k))$ for $\mathit{Cons}(n_k)$. The caveat in this approach is that function fitting is difficult over large data sets due to both the number of parameters involved and due to the presence of discontinuities. We tackle this problem as follows:

- We function fit for C, starting at $\mathit{Term}(n_k)$, progressively conjoining terms $\mathit{Term}(n_i)$ for $i = k-1, k-2, \ldots, 1$, stopping when we find a smooth function. This reduces the number of unit vectors we consider and by the Sufficiency of Witnesses considers the smaller number of data points.
- We reduce the number of system parameters for function fitting using treatment learning. For C, we use the data seen during system testing to find the subset $I_n \subseteq I$ of system parameters that most affect the values of the unit parameters in $\mathit{Vars}(C)$.

For all terms in $\mathit{Cons}(n_k)$ that are not considered in a given iteration of function fitting, i.e., terms in $\mathit{Cons}(n_k)$ but not in C, we use treatment learning to find satisfying assignments. By the Monotonicity of Witnesses, we have more data points to cover these terms than to cover $\mathit{Cons}(n_k)$, increasing the likelihood of finding good treatments.

Algorithm. We now describe $Cover$, our coverage algorithm presented in Algorithm 1. The algorithm works as follows:

1. *Lines 2–4*. We perform n-factor combinatorial Monte Carlo (MC) simulations by picking values over a space sp; a d-dimensional space for the d input parameters constrained by their data types. Unlike traditional random MC, n-factor MC generates test cases such that every possible combination of input parameters equal to size n appears at least once in the test suite [22]. For every system vector a, we monitor the unit and capture the unit vector b together with the path constraint for the path taken within the unit. The set of path constraints are summarized in T; system and unit vectors are stored in sets V and v.
2. *Lines 7–11*. We traverse the nodes in T in breadth first order. The treatment learner learns a treatment for each node n in T as long as its sibling is also covered. Since the treatment learner is a contrast set learner, it can be used to identify a set $I_n \subseteq I$ and ranges R_n of parameters in I_n, only when given data points that differentiate n from its sibling.
3. *Lines 13–16*. For each satisfiable node n in T not covered by MC simulations, we store the assignment $\boldsymbol{i}$ satisfying $\mathit{Cons}(n)$. We start with a constraint C set to $\mathit{Term}(n)$ and progressively strengthen C until we find a system vector to cover n. As we want to fit a function that maps I to i, we keep track of the parameters in C in i_n and the restriction of $\boldsymbol{i}$ to the parameters i_n in $\boldsymbol{i_n}$. The function $ComputeMap$ finds a function f_n such that $I_n = f_n(i_n)$ using function fitting.
4. *Lines 17–19*. We iterate over all satisfiable nodes n in T not covered during system testing. For each such n we run a system level test by composing a system vector as follows: (a) take $\boldsymbol{I_n} = f_n(\boldsymbol{i_n})$ such that it is consistent with the ranges r_j for all $j \in I_n$ as returned by the treatment learner in Line 10 and (b) for all other system level parameters $j \in I \setminus I_n$, pick a value from the ranges r_j returned by the treatment learner in Line 10.

Algorithm 1. $Cover(S, U)$

```
input : System $S$ with inputs $I$ with $d = |I|$, unit $U$ with inputs $i$
1  $sp \leftarrow \mathbb{R}^d$;
2  Perform $n$-factor combinatorial MC simulations over space $sp$;
3  $(V, v) \leftarrow \{(a, b) \mid a$ is a system level vector and $b$ is the corresponding monitored
   unit level vector$\}$;
4  $T \leftarrow (PC$ from $U)$;
5  repeat
6      $T' \leftarrow T$;
       // Do $BFS$ on $T$
7      for (node $n$ in $T$ using BFS) do
8          if ($n$ and $n$'s sibling are covered) then
               // Use contrasting data to learn a treatment
9              $V' \leftarrow \{a \in V \mid a$ covers $n\}$ and $V'' \leftarrow V \setminus V'$;
10             $(I_n, R_n, \_) \leftarrow RunTAR3(I, V, V', V'')$;
11             $\forall j \in I_n$ store the range $r_j \in R_n$ for $j$;
12         else
13             if ($n$ is satisfiable but not covered) then
                   // Compute $f_n$ such that $I_n = f_n(i_n)$
14                 $\boldsymbol{i} \leftarrow$ model for $Cons(n)$;
15                 $C \leftarrow Term(n)$;
16                 $(I_n, i_n, f_n) \leftarrow ComputeMap(C, I, V, v, n, Parent(n), \boldsymbol{i})$;

       // Build new test-cases
17     for ($n$ in $T$ satisfiable but not covered) do
18         Run $S$ with a consistent valuation using $f_n(\boldsymbol{i_n})$ and $\forall j \in I \setminus I_n$ using $r_j$
           from Line 10;
19         $T' \leftarrow T' \cup (PC$ from $U)$;
20     $T \leftarrow T'$;
21 until ($T$ has no unprocessed nodes);
```

The function fitting algorithm $ComputeMap$, shown in Algorithm 2, works as follows:

1. *Lines 1–4* We compute i_n occurring in C and the restriction of the model $\boldsymbol{i}$, for $Cons(n)$, to i_n. We use treatment learning to isolate a set $I_n \subseteq I$ most likely to affect i_n and to determine if the data points in V and v have a smooth relationship.
2. *Lines 5–6* If the relationship is smooth we build the map f_n such that $I_n = f_n(i_n)$.
3. *Lines 8–10* If the relationship is not smooth, we strengthen C by including the parent term from $Cons(n)$ and then recursively call $ComputeMap$.
4. *Lines 12–22* If we cannot find a smooth relationship by including all terms in $Cons(n)$, then we use the Sufficiency of Witnesses to walk up the parent hierarchy of n to reach a node n'' that has at least Threshold data points that witness covering n''. By Assumption 2, we have at least one path that was taken through the unit during system testing. If we find two data points that covered a node in the parent hierarchy of n, we attempt a linear fit and return. If we cannot find at least two data points, we run more MC simulations.

Algorithm 2. $ComputeMap(C, I, V, v, n, n', \boldsymbol{i})$

```
input : Constraint $C$ such that $Cons(n_k) \Rightarrow C$, system inputs $I$, system vectors $V$,
        unit vectors $v$, a node $n$ that we want to cover, a node $n'$ that is in the parent
        hierarchy of $n$ and a model $\boldsymbol{i}$ for $Cons(n)$
output: $(I_n, i_n, f_n)$ where $I_n = f_n(i_n)$ and $i_n = Vars(C)$)
1  $i_n \leftarrow Vars(C)$;
2  $\boldsymbol{i_n} \leftarrow$ restriction of $\boldsymbol{i}$ to $i_n$;
   // Find a subset of $I$ for function fitting
3  $V' \leftarrow \{a \in V \mid a$ is in 20% of points closest to $Cons(n)\}$ and $V'' \leftarrow V \setminus V'$;
4  $(I_n, R_n, smooth) \leftarrow RunTAR3(I, V, V', V'')$;
5  if (smooth) then
6      Build map $I_n = f_n(i_n)$;
7  else
       // Strengthen constraint and try again
8      if ($n'$ exists) then
9          $C \leftarrow C \wedge Term(n')$;
10         $(I_n, i_n, f_n) \leftarrow ComputeMap(C, I, V, v, n, parent(n'), \boldsymbol{i})$;
11     else
           // If no smooth relation between $I_n$ and $i_n$, then
              walk up the parent of $n$, pick a node with
              Threshold points, and attempt a linear fit
12         $n'' \leftarrow n$;
13         while ($Parent(n'')$ exists) do
14             $C \leftarrow C \wedge Term(Parent(n''))$;
15             $n'' \leftarrow Parent(n'')$;
16             $V' \leftarrow \{a \in V \mid a$ covers $n''\}$;
17             if ($|V'| \geq Threshold$) then
18                 break;
19         $V'' \leftarrow V \setminus V'$;
20         $(I_n, R_n, \_) \leftarrow RunTAR3(I, V, V', V'')$;
21         $i_n \leftarrow Vars(C)$;
22         Build map $I_n = f_n(i_n)$;
```

We use the treatment learning algorithm TAR3, presented in Algorithm 3 for the following two purposes in our coverage algorithm:

Learning Rules for Covered Nodes. We use TAR3 to determine the subset of system inputs and their ranges that covered nodes at the unit level. For every node n that was covered during system testing, if its sibling was also covered, then we have a partition of the data points at the system level into one set that covered n and the other set that covered its sibling. We use TAR3 with these partitions to learn rules that will either visit n or its sibling; Line 10 of Algorithm 1. We use these rules at Line 18 to pick values for a subset of I as described in the algorithm.

Algorithm 3. $RunTAR3(I, V, V', V'')$

```
input : System level parameters $I$, system level vectors $V$ and contrast sets $V' \subset V$
        and $V'' = V \setminus V'$.
output: $(I', R, smooth)$ where $I' \subseteq I$, $R$ is a set of ranges for each parameter in $I$,
        $smooth$ is set to true by examining the output
1 Call TAR3 with $V$, $V'$ and $V''$;
2 Compose $I' \subseteq I$, $R$ and $smooth$ based on the results of running TAR3;
3 Return $(I', R, smooth)$;
```

Learning Inputs for Function Fitting. We attempt to fit a function to cover node n using a weak C initially set to $Term(n)$. This C is progressively strengthened as seen in Algorithm 2. For each C, we construct contrast sets by partitioning the data points into a.) the 20% of the data points nearest in Euclidean distance to the PC boundary and b.) all remaining points. These sets are used to learn a small subset of I most influencing i close to the PC boundary. We use this reduced subset of I for function fitting.

As an example, in Figure 4 the desired $\boldsymbol{i}$ are represented by the gray rectangle in the center of the plot. Curves are built from data pairs seen during program execution; dotted circles surround the data nearest the PC boundary and comprise a contrast set. TAR3 returns the I that most affect the i near the PC boundary. We also use TAR3 to determine whether a smooth relationship exists between subsets of i and I. In Figure 4, the relationship between i and I appears to be discontinuous. To each side of the PC boundary, a small variation in system values leads to a large variation in the unit values; it is possible to get two different unit values for the same system level value.

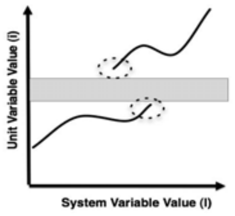


Fig. 4. A non-smooth relationship between a system and a unit parameter. The gray region represents values of $\boldsymbol{i}$ not seen during testing. Dotted circles surround data closest to the boundary.

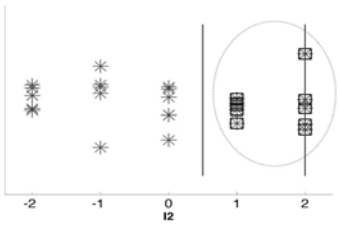


Fig. 5. Bars outline a rule that guides execution through Node 2. Data points (asterisks) are boxed if the runs pass through Node 2. The dotted oval outlines a contiguous region that suggests f_2 is smooth.

Discussion. We now discuss the assumptions made in our coverage algorithm and also the conditions under which the algorithm makes progress. We make the following assumptions in our coverage algorithm:

1. The unit U can be analyzed using concolic execution,
2. At least one path in U is taken during system testing.

The first assumption is required since our goal is to use unit level concolic execution to improve system testing. The second assumption may be satisfied using one of the following two approaches:

1. Iteratively choose smaller systems that enclose U, until we find a system such that at least one path is taken in U during system testing.
2. Pick the earliest method U' up the call chain of U that has at least one path covered during system testing and then run $Cover(S, U')$. This increases the test vectors that explore U' and hence the likelihood of taking paths in U.

We remark that by using a breadth first exploration of the constraints tree, we ensure that when we attempt to cover a node, all its parent nodes have been processed. This ensures that when we build a system level vector for a node n, we have learnt ranges for all nodes in its parent hierarchy; the system level vector is composed using these ranges and the function f_n.

Remark 1 (Progress). In the presence of perfect function fitting, if we have an over-approximation of the subset of I_n that affect the $i_n = Vars(Cons(n))$ for every node n that is satisfiable at the system level, then the algorithm will eventually cover n.

Consider a satisfiable node n that cannot be covered by considering any constraint weaker than $Cons(n)$. As we strengthen the C from $Term(n)$ to $Cons(n)$, we eventually include in C all terms from $Cons(n)$ and all i_n in $Vars(Cons(n))$. If we find a perfect function f, such that $I_n = f(i_n)$, and if I_n includes all the I that affect i_n, we are guaranteed to cover n. We use TAR3 to extract I_n. We can supplant TAR3 with static analysis techniques, such as [23], to learn an over approximation of the set I_n. Note that due to loops or recursion, our algorithm may not terminate.

5 Experience

In this section, we present our experience using the technique proposed in this paper on several examples. Two of these examples are purely illustrative, the third is a classic aerospace example. Planned experiments include larger aerospace examples: flight control software for unmanned aerial vehicles and a prototype conflict detection and resolution algorithm.

Our algorithms are implemented in the context of analyzing C code. We use MATLAB scripts to generate an initial suite of system vectors V given the known I, and to execute programs instrumented for concolic execution. The concolic execution framework is implemented using CIL [24], the C Intermediate Language, that provides an API for the analysis of C programs, to instrument user code. We use CIL to walk the intermediate representation of the program and insert calls to a set of runtime listeners. The user program is then re-generated from the intermediate representation, linked with our runtime library and run. During MC simulations, we use the instrumented version of the unit to monitor unit and system inputs and to capture paths that were taken within the unit. The constraints tree generated during MC simulations is used as an input to a subsequent solve cycle, where we solve for paths not taken within the unit during system level testing, replay solutions found and thus explore the tree to completion; we solve path constraints using Yices [25]. The outputs of these steps are a fully explored constraints tree T together with models for all satisfiable paths, a set of unit vectors v and the corresponding system vectors V that we monitored during MC simulations. These outputs are fed to MATLAB scripts that use I, T, i, V and v to perform treatment learning and function fitting, and to predict new $\boldsymbol{I}$ that better cover T in subsequent iterations. Two steps in our current process are manual, and we have plans to automate both: a) determining whether TAR3's treatments suggest smooth functions, and b) choosing whether to begin execution of the new $\boldsymbol{I}$.

A Piecewise Linear Case Study. We will first use the simple, piecewise linear implementation in Program 1. Although the f_n for this example can be found by hand or by symbolic execution, we use it here to illustrate our technique. *Unit* is instrumented to perform concolic execution and graphical results are shown in Figure 3. All invocations of Unit begin at Node 1 in Figure 3. Control flow from Node 1 is determined by f_1, which is $i1 = I2$. If $I2 > 0$, control flow passes to Node 2; otherwise, to Node 5. For demonstration, we treat f_1 as unknown, and determine it using our heuristic methods.

We initially create 25 test cases using values for $I1$ and $I2$ between -2 and 2 (Algorithm 1, Lines 2–4). Nodes 4 and 7 within Unit are not covered; concolic execution provides the unit input constraints that will cover them. Figure 2, Lines 2–4 give the required unit level parameters: g2, g1, and i1. Lines 6–11 show T for Unit; Line 11 corresponds to Node 7, and has an 'S' to show that the constraint is satisfiable at the unit level.

The generated constraint tree is traversed using breadth-first search (Algorithm 1, Lines 7–16). Lines 6 and 9 in Figure 2 indicate covered sibling nodes (Algorithm 1, Line 8); TAR3 automatically returns the rule set for passing through Node 2, ($0.5 \leq I2 \leq 2$), as shown by parallel bars in Figure 5. Similarly, TAR3 discovers ($-2 \leq I2 \leq 0.5$) for passing through Node 5. Note that TAR3 does not capture the exact location of the constraint boundary between Nodes 2 and 5. TAR3 can not learn system constraints for Nodes 3 and 6 as there is no contrasting data.

TAR3 is then used to reduce the subset of values of I_n for function fitting. Contrast data sets are built by isolating the 20% of unit input data nearest the constraint boundary. For Node 4, TAR3 suggests that $g2$ depends on a smooth relationship involving only $I1$. To cover Node 7, our approach first considers all data satisfying the weakest constraint ($g1 \leq -3$); TAR3's results are in Figure 6. The data nearest in value to the constraint boundary are spread discontinuously across I1 and I2 space. TAR3 makes a prediction involving a subset of the points. This happens when the the relationship between i and I is not smooth; in this case, the relationship between $g1$ and $I2$ has a discontinuity at $I1 = 0$. The constraint is strengthened by considering the data satisfying $i1 \leq 0 \wedge g1 \leq -3$. By the Monotonicity of Witnesses, this yields fewer data points; there are a total of 15 data points passing through Node 5. TAR3 now suggests there is a smooth $g1 = f_7(I1, I2)$.

For Node 7 the exact solution $g1 = I2$ is predicted using function fitting (Algorithm 2), with an error less than 10^{-15}. For Node 4 the solution $g2 = I1 + 3$ is predicted with an error less than 10^{-14}. These approximations, along with the previously discovered system level constraints (Algorithm 1, Line 10), enable building new test inputs for $I1$ and $I2$ to cover Nodes 4 and 7 on the next test iteration (Algorithm 1, Lines 17–19).

Program 2. The System Function in the Prototype Quadratic Example. The Unit Function is the same as in Program 1, except that the Unit Function for this case expects inputs of type *double.*

```
double g1=1.0, g2=2.0;
int System(double I1, double I2)
{
  if (I1 > 0) g1 = I2;
  else g1 = -I2;
  g2 = I1*I2+3.0*I1*I1+I2*I2;
  Unit(I2, I1);
}
```

A Piecewise Quadratic Case Study. As a simple example of how our technique could be used in the presence of nonlinear constraints (that are not typically handled by off-the-shelf solvers), we propose the example in Program 2. Program 2 and Program 1 differ in the use of *doubles* instead of *ints* and the nonlinear assignment formula for $g2$ before *Unit* is called. T is identical to the one given in Figure 2 and Figure 3. A breadth-first search over covered nodes gives identical results to the previous section.

TAR3's results for Node 4 are shown in Figure 7. The treatment was unable to bound all of the contiguous boxed data; this suggests that f_4 is smooth but nonlinear.

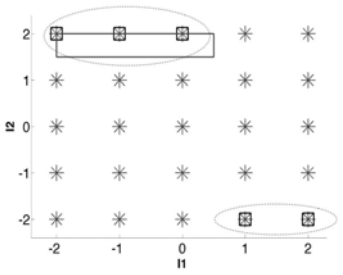


Fig. 6. Node 7's treatment

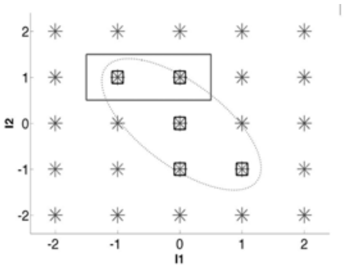


Fig. 7. Node 4's treatment

Function fitting is applied for Nodes 4 and 7. Node 7's results are identical to those in the previous section. For Node 4, function fitting gives a residual error of less than 10^{-15} and the exact solution $g2 = 3.0 * I1^2 + I2^2 + I1 * I2$. Our algorithm first attempts to create an $\boldsymbol{I}$ that satisfies $g2 \leq 0$ and is consistent with the system parameters and ranges learned previously (Line 10 of Algorithm 1), but discovers that there is an inconsistency. There are no real roots that satisfy the constraint for $g2$ given f_4 and the range constraints for Node 4's parent (Node 2). Function fitting for Node 2 yields the exact result $i1 = I2$. By simple substitution the correct system constraint is $I2 > 0$. An examination of Node 4's constraint reveals that the two system constraints are unsatisfiable; no system test leads us to Node 4.

An Aerodynamics Case Study. In this aerodynamics case study the code predicts the drag coefficient C_d, as calculated by the USAF Stability and Control DATCOM manual [26]; it can be found at `https://c3.nasa.gov/dashlink/projects/57/#c0`. C_d is used in the yaw control law for a supersonic aircraft designed to fly between 30,000 and 80,000 feet at Mach numbers M between 0.8 and 3.0. M is a ratio of the plane's airspeed to the speed of sound, and is calculated by measuring two different pressures, P_t and P_s. The system I consists of three arguments from sensors: P_t, P_s, and the altitude Alt. This sensed data is used to calculate M, compressible and incompressible skin friction coefficients C_f and C_{fb}, and the corresponding terminal skin friction coefficients C_fT and $C_{fb}T$. For subsonic ($M < 1$) compressible flow in air, M is given by Equation 1; for supersonic ($M >= 1$) flow, M is found implicitly using the *Rayleigh Pitot tube formula* [27], shown here as Equation 2.

$$M = \sqrt{5\left[\left(\frac{P_t}{P_s}\right)^{\frac{0.4}{1.4}} - 1\right]} \quad (1) \qquad \frac{P_t}{P_s} = \left(\frac{5.76M^2}{5.6M^2 - 0.8}\right)^{3.5} \frac{2.8M^2 - 0.4}{2.4} \quad (2)$$

For Equation 2, there is *no explicit formula* for M given P_t, P_s. One code component uses Newton's Method to solve Equation 2, and is used as a black box for our technique. C_f, C_{fb}, C_fT and $C_{fb}T$ are complicated nonlinear functions of M and Alt [26]. The unit calculates C_d based on the skin friction and the base drag. The relationships between C_d and the unit inputs are nonlinear, but the constraints defining the relationships are linear and easy to both discover and solve using concolic execution techniques.

```
  [Parameters]
2 CfbT
3 Cf
4 M
5 CfT
6 Cfb
  [Tree]
8  (Cf > CfT) (C)
9     (M >= (780000 / 1000000)) (C)
10      (M > (1040000 / 1000000)) (C)
11        (M >= (600000 / 1000000)) (C)
12          (Cfb > CfbT) (C)
13            (M >= 1) (C)
14              (M <= (2000000 / 1000000)) (C)
15              (M > (2000000 / 1000000)) (C)
16            (M < 1) (S)
17          (Cfb <= CfbT) (S)
18        (M < (600000 / 1000000)) (S)
19      (M <= (1040000 / 1000000)) (S)
20    (M < (780000 / 1000000)) (S)
21  (Cf <= CfT) (S)
```

Fig. 8. The constraints tree after seven rounds of initial testing

We begin our testing of the system by looking at nominal ranges for the aircraft: Alt between 30 and 80 thousand feet, P_t between 0.0145 and 25, and P_s between 0.00971 and 3.5. Performing 2-factor combinatorial testing [28] with 5 bins for each of these parameters gives 9 initial test cases. Two of these cases have $P_t < P_s$, a physical impossibility, and are thrown out.

The constraints tree T for our 7 initial test cases covers only 2 paths through the tree, as shown in Figure 8. T is traversed using a breadth-first search. For the nodes at lines 21 and 17 of Figure 8, TAR3 suggests a smooth relationship between the unit parameters and the system parameters P_s and Alt. For the nodes at lines 16 and 18-20, TAR3 suggests a smooth relationship between M and the system parameters P_t and P_s. Function fitting is performed for the nodes not covered by system testing, using all 7 initial data points, giving the approximation $M = 5.7022 + 0.0035 * P_t^2 - 0.0092 * P_s * P_t + 0.7255 * P_s^2 - 0.0124 * P_t - 3.4665 * P_s$ with a residual of 0.0479. This process is repeated to find approximations between the unit parameters C_f, C_{fb}, C_fT and $C_{fb}T$, and the system parameters P_s and Alt that were implicated by TAR3.

Constraint solving is then used to find test inputs for each node not covered in T. The result is 17 new $\boldsymbol{I}$, which are used for new simulations. Concolic execution records the paths taken through the unit; the resulting T has 5 covered paths with 21 covered nodes and 12 nodes not covered—only 5 of the nodes not covered are satisfiable. When the new T is compared against the one in Figure 8; the constraints at lines 17, 19 and 21 are covered. After two rounds of testing, our method uses 24 tests to illuminate a constraints tree with 21 covered nodes and 12 nodes not covered.

We compared our technique against state-of-the-art black box testing by generating a test suite with 25 n-factor combinatorial tests; n-factor combinatorial testing typically obtains better coverage than random Monte Carlo testing [22,29]. With a comparable number of tests (24 vs. 25) our technique achieves significantly higher coverage (21 covered nodes) than the coverage obtained by n-factor combinatorial testing alone (16 covered nodes).

6 Conclusion

We described a testing technique that combines the strengths of black-box system simulation with white-box unit symbolic execution to overcome their weaknesses. The technique uses machine learning, function fitting and constraint solving to iteratively guide the generation of system-level inputs and increases the testing coverage. We showed in the experience section that we could use our tool to increase coverage of a unit using fewer test cases compared to state-of-the-art combinatorial testing. System level simulation can be expensive, and using information from white-box techniques allowed us to significantly decrease the time cost. White-box techniques, like concolic execution, may not scale to a full system. This is especially true when the system either contains non-linear components or contains components for which the source code is unavailable. Covering each white-box unit separately is an option, but there are likely to be test cases which are not possible given the constraints of the full system. As an example, the values of the Mach number and the friction coefficients in our aerodynamics case study are constrained by the measured values of the pressures P_t and P_s and the altitude. This means that, even though the Mach number and the friction coefficients are treated as independent inputs to our unit, the values of these variables cannot truly vary independently. If we performed only unit-level full coverage, we may miss dead code that is unreachable given the system, or we may spend too much time exploring behaviors in the unit that are not possible given the unit's true calling context. In the future, we plan to study alternative approaches to machine learning (e.g. Daikon) and to perform a thorough evaluation of the technique to determine its utility in practice.

References

1. Acevedo, A., Arnold, J., Othon, W., Berndt, J.: ANTARES: Spacecraft simulation for multiple user communities and facilities. In: AIAA 2007–6888 Mod. and Sim. (2007)
2. Pasareanu, C., Mehlitz, P., Bushnell, D., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA (2008)
3. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: FSE (2005)
4. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. ACM (2005)
5. King, J.C.: Symbolic execution and program testing. CACM (1976)
6. Menzies, T., Hu, Y.: Data mining for very busy people. IEEE Computer (2003)
7. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
8. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM CCS (2006)
9. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: FSE (2006)
10. Sen, K.: Concolic testing. In: ASE (2007)
11. Gay, G., Menzies, T., Davies, M., Gundy-Burlet, K.: Automatically finding the control variables for complex system behavior. In: ASE (2010)
12. Bay, S., Pazzani, M.: Detecting change in categorical data: Mining contrast sets. In: KDDM (1999)
13. Agrawal, R., Imeilinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD (1993)
14. Cai, C., Fu, A., Cheng, C., Kwong, W.: Mining association rules with weighted items. In: IDEAS (1998)
15. Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Machine Learning 11 (1993)
16. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artificial Intelligence (1997)
17. Trefethen, L.N., David Bau, I.: Numerical linear algebra. SIAM (1997)
18. Strang, G.: Linear algebra and its applications, 3rd edn. Thomson Learning (1988)
19. Burden, R.L., Faires, J.D.: Numerical analysis, 7th edn. Brooks/Cole (2001)
20. Bartle, R.: The elements of real analysis, 2nd edn. John Wiley & Sons (1976)
21. Schumaker, L.L.: Spline functions: basic theory. Wiley Interscience (1981)
22. Cohen, D., Dalal, S., Parelius, J., Patton, G.: The combinatorial design approach to automatic test generation. IEEE Software 13, 83–88 (1996)
23. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA (2007)
24. Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: International Conference on Compiler Construction, pp. 213–228 (2002)
25. Dutertre, B., Moura, L.D.: The YICES SMT solver. Technical report, SRI International (2006)
26. Finck, R.: USAF stability and control DATCOM. Technical Report AFWAL-TR-83-3048, USAF (1978)
27. Anderson, J.D.: Fundamentals of Aerodynamics, 3rd edn. Mc-Graw Hill (2001)
28. Gundy-Burlet, K., Schumann, J., Barrett, T., Menzies, T.: Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In: AIAA Aerospace (2009)
29. Dunietz, I., Ehrlich, W., Szablak, B., Mallows, C., Iannino, A.: Applying design of experiments to software testing: experience report. In: ICSE, pp. 205–215 (1997)

Infeasible Code Detection

Cristiano Bertolini[1], Martin Schäf[1], and Pascal Schweitzer[2]

[1] United Nations University, IIST, Macau
[2] Australian National University

Abstract. A piece of code in a computer program is infeasible if it cannot be part of any normally-terminating execution of the program. We develop an algorithm for the automatic detection of all infeasible code in a program. We first translate the task of determining all infeasible code into the problem of finding all statements that can be covered by a feasible path. We prove that in order to identify all coverable statements, it is sufficient to find all coverable statements within a certain minimal subset. For this, our algorithm repeatedly queries an oracle, asking for the infeasibility of specific sets of control-flow paths.

We present a sound implementation of the proposed algorithm on top of the Boogie program verifier utilizing a theorem prover to provide the oracle required by the algorithm. We show experimentally a drastic decrease in the number of theorem prover queries compared to existing approaches, resulting in an overall speedup of the entire computation.

1 Introduction

Static analysis allows us to detect undesired behavior of a program before it is executed or even compiled. A particular application of static analysis is to identify code fragments that show *only* undesired behavior. Tools implementing this approach detect code which is never part of an execution that terminates normally. We refer to this type of code as *infeasible code*. The terminology infeasible code is justified as follows: an execution is infeasible if it does not terminate normally (note that we do not model error states and thus, any terminating execution terminates normally). A path is infeasible if all its executions are infeasible. And code is infeasible if it only occurs on infeasible paths. Compared to unreachable code, where no feasible path *ends* in the considered code, infeasible code is a more general concept as it only requires that no feasible path *contains* the considered code.

Subsets of infeasible code are, for example, found by the static analyzers in modern development environments such as Eclipse. These tools detect simple, yet common errors such as guaranteed null pointer dereference, use of uninitialized variables, or unreachable program fragments. In practice, finding such type of errors is one of the most frequent applications of static analysis. Thus, improving the detection of infeasible code is a worthwhile problem.

An intriguing property of infeasible code is that it can be detected without prior knowledge of the desired program behavior or it's environment (e.g., the possible input values). If a piece of code is infeasible, it will stay infeasible even if the context is subsequently restricted by other means such as adding guarding statements or specifying admissible input values. Thus, infeasible code can be detected while typing the program, and infeasible code can only be eliminated by changing the code fragment itself and not by changing other code.

Recently, new static analysis approaches have emerged that use formal methods to prove the presence of infeasible code [11, 14]. They prove a particular program statement to be infeasible code by encoding all paths passing through the statement in a logic formula. The formula is satisfiable if there exists a normally terminating execution of the program following one of these paths. The benefit of proving the presence of infeasible code with this approach is that it does not produce false warnings [12].

To detect *all* infeasible code in a program, this method is repeatedly applied to different program statements. Each application invokes a theorem prover query and is thus computationally expensive. It is possible to query the infeasibility of several statements simultaneously, in order to reduce the number of queries. Minimizing the number of queries can help us to devise more efficient ways to process the entire code, however, using more complex queries may be computationally more expensive. Thus the following question arises:

What is an efficient strategy to detect all infeasible statements in a given program?

In this paper, we show that the problem of identifying all infeasible code can be expressed as a set cover problem on a minimal subset of program statements. We then show that the problem of detecting infeasible code is equivalent to proving the non-existence of a feasible path cover for this subset of statements in the control-flow graph of the program. We further show that a feasible path cover of this set covers all feasible statements in the program. This in particular shows how the feasibility or infeasibility of every statements can be determined from the feasibility information on the subset of statements. We present a query optimal greedy algorithm to compute a feasible path cover and a sound implementation. The implementation uses a theorem prover (we use Z3 [6]) as an oracle to check the existence of a feasible control-flow path in a particular set. We show experimentally that the proposed method is more efficient in terms of oracle calls and computation time than existing approaches.

Related Work. Numerous approaches exist that, among other things, show infeasible code to be faulty. E.g., when a test case executes infeasible code it must reveal an error. Many of these approaches suffer from false warnings or require a strong user interaction (e.g., [8, 19]). Moreover, the approaches do not prove code to be infeasible. We restrict the discussion to static error detection approaches that detect statements from which a good state is unreachable. Findbugs [13] detects contradicting control-flow using pattern matching. It detects statements similar to infeasible code, however, pattern matching is neither sound nor complete.

Encoding the feasibility of control-flow paths as a logic formula goes back to the idea of predicate transformers [7]. An algorithm that uses formal methods to prove that a statement cannot be *reached* by a feasible execution is presented in [14]. Unreachable code is a special type of infeasible code. An algorithm to detect if a particular control location is never passed by a feasible execution is developed in [11]. They query a theorem prover whether there exists some feasible path containing a particular location. Their approach detects doomed locations while our approach detects infeasible statements. By inserting auxiliary locations, the two approaches can be reduced to each other. However, one of the central insights of this paper is that, in order to find all statements occurring on feasible paths, it is in general *not* sufficient to check only control locations.

In [12] the algorithm from [11] is extended to an algorithm that detects all such locations. They present a strategy to minimize the number of theorem prover queries to detect infeasible code on a loop-free abstraction of the program. In this paper, we present a more general type of query. This type of query allows us to check infeasibility of several statements simultaneously. Moreover, we prove that it is possible to determine all infeasible statements by only checking a minimal subset of all statements. Without the use of auxiliary locations, our proof *cannot* be translated to suit the approach using locations. Any direct proof for program locations seems to require further properties of the program. We show our approach, which also works for programs with loops, further reduces the number of queries and prove that our approach is query optimal.

Algorithms to compute feasible path covers or sets of infeasible control-flow paths have been proposed in software testing (e.g. [4]). These approaches either require an executable program, or over-approximate the feasible path cover.

Organization of the paper. In Section 2 we give some examples of infeasible code. In Section 3 we formalize the notion of infeasible code and show that it's detection can be expressed as a path cover problem. In Section 4 we then present an algorithm to check the existence of a feasible path cover of a control-flow graph that uses an oracle to check if a given set of paths is infeasible. In Section 5 we show how this oracle can be realized using weakest liberal preconditions. In Section 6 we present a prototype implementation of our algorithm and in Section 7 we experimentally compare this implementation with existing implementations in terms of theorem prover calls and computation time.

2 Examples of Infeasible Code

Figure 1 provides 4 example programs that demonstrate the usefulness of infeasible code detection. In `ex01`, line 3 is infeasible because any execution passing that line will loop forever. This example shows that infeasible code also refers to code that does neither reach a normal terminating state nor violates an assertion. However, infeasible code detection only detects non-termination if any execution entering the loop must run forever.

The program `ex02` has infeasible code in line 3. It shows that code can be infeasible because its execution always causes an error at some later stage. Indeed,

```
1 void ex01(int x) {
2   while (x>0) {
3     x=(x/2)+1;
4   }
5 }
```

```
1 void ex02(C a, int x) {
2   if (a==null)
3     x=-1;
4   a.toString();
5 }
```

```
1 void ex03() {
2   int a, b;
3   a=1;
4   if (a>0) b=1;
5   a=b;
6 }
```

```
1 void ex04(C a) {
2   int y=0;
3   if (a!=null)
4     y=1;
5   if (y==0)
6     a.toString();
7 }
```

Fig. 1. The example programs show different possible causes for code to become infeasible

on any execution passing line 3, the reference `a` is guaranteed to be `null` and thus, the program terminates abnormally in line 4. Note that the only infeasible statement in this program is line 3. All other statements are part of feasible executions.

In practice, we use automatically generated assertions to guard pointer dereferences and other properties. In general, the approach presented in this paper can be used with arbitrary safety properties. E.g., we can use infeasible code detection for definite-assignment analysis and encode the property that every variable must be written once before it is read by using helper variables and assertions. To this end, we can show that in `ex03`, the variable `b` is initialized on any feasible path. In contrast, the Java compiler rejects this program claiming `b` might be not initialized if `a` is not positive at line 4. We can encode other properties, such as array-bound checking, or locking behavior in the same way.

A more complex example of infeasible code is given in `ex04`. Any path containing line 6 is either infeasible because the conditional evaluates to false, or `null` is dereferenced. For code to be infeasible, it is not necessary that all paths are infeasible for the same reason or diverge at the same control location.

3 Infeasible Code, Effectual Sets, and Path Covers

A program is defined by a control-flow graph $\mathcal{P} = (S, \delta, \Sigma)$. A control-flow graph is a connected directed graph where S is the set of control locations and Σ is the set of instructions in the program. A program statement $\mathit{st} = (s, \mathit{inst}, s') \in \delta$ is an instruction $\mathit{inst} \in \Sigma$ at a control-location s whose execution ends in a control-location s'. The transition relation $\delta \subseteq S \times \Sigma \times S$ represents the set of statements in $\mathcal{P}$. W.l.o.g., we assume that the program has a unique source and a unique sink node. A path from a node s_1 to a node s_{k+1} in $\mathcal{P}$ is a sequence of statements $\pi = \mathit{st}_1 \dots \mathit{st}_k = (s_1, \mathit{inst}_1, s_2) \dots (s_k, \mathit{inst}_k, s_{k+1})$, s.t. $\mathit{st}_1, \dots, \mathit{st}_k \in \delta$. Note that, as customary in the context of control flow graphs, a path may use vertices repeatedly. A *complete* path is a path that starts in the source node and ends in the sink node of the graph. Throughout this paper, unless stated otherwise, the word path always refers to a complete path.

Infeasibility. We assume that the semantics of a statement st is given by a weakest liberal precondition operator $P = wlp(st, Q)$, s.t. any execution of st starting in a state satisfying P results in a state satisfying Q or does not terminate normally [7]. We say an execution does not terminate normally if it blocks, either because a conditional statement is not satisfied or it crashes because an (possibly implicitly) assertion is violated, or it runs forever. We extend the weakest liberal precondition from statements to paths in the obvious way.

Definition 1. *Given a program $\mathcal{P} = (S, \delta, \Sigma)$, a path π in $\mathcal{P}$ is infeasible if $wlp(\pi, false) = true$.*

Here, *true* denotes the set of all possible states and $false$ the empty set of states. Note that we do not take into account the reason for paths being infeasible. We are only interested in the fact that their executions do not terminate normally.

In general, not every control-flow path in a genuine program is feasible. E.g., control-flow paths may be infeasible because of complex conditional branching. Only if a statement is not part of any feasible execution we call it infeasible code.

Definition 2. *Given a program $\mathcal{P} = (S, \delta, \Sigma)$, a statement $st \in \delta$ is infeasible code, if there is no feasible path π in $\mathcal{P}$ that contains st.*

There are two reasons why a statement can be infeasible code. One is that it is not part of any (terminating) execution, the other is that it is only part of executions that terminate in an error state.

Effectual sets. For the detection of infeasible code, it is not necessary to consider all edges (statements) in a control-flow graph. It suffices to focus on a minimal subset of edges of which each control-flow path contains at least one. As shown in [3, 12], this set can be identified using a partial order over control-flow edges. The minimal subset can be used to decide whether there is infeasible code. However, for our purpose of determining *all* infeasible code, there are examples of control flow graphs where these sets are not sufficient (see [2] for an example).

Definition 3. *Given a program $\mathcal{P}$ and two statements $st, st' \in \delta$, we write $st \preceq st'$ if every complete or infinite path π in $\mathcal{P}$ that contains st also contains st'.*

We remark that in acyclic graphs the defined relation coincides with the one used in [12]. The relation $\preceq$ is reflexive and transitive, therefore the relation $\simeq = \preceq \cap \preceq^{-1}$ is an equivalence relation. We denote by $[st]$ the equivalence class of a statement st under $\simeq$. We say that $[st] \preceq [st']$ if and only if $st \preceq st'$.

For a set $\delta' \subseteq \delta$ we define $\text{cov}(\delta')$ to be the maximal number of elements of δ' contained in a (not-necessarily feasible) path.

We call a set $\delta' \subseteq \delta$ *effectual* if it is a maximal set of statements that are all minimal w.r.t. $\preceq$, such that for any two distinct statements $st, st' \in \delta'$ we have $st \notin [st']$. We will usually denote effectual sets by $\mathcal{F}(\delta)$.

Path covers. A *path cover* of the program $\mathcal{P} = (S, \delta, \Sigma)$ is a set of paths such that each statement in δ is contained in at least one of the paths. A path cover is feasible if it contains only feasible paths. Path covers and effectual sets are the key element to our method of determining infeasible code.

Theorem 1. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program and $\mathcal{F}(\delta) \subseteq \delta$ an effectual set. Program $\mathcal{P}$ has no infeasible code, if and only if there is a feasible path cover of $\mathcal{F}(\delta)$.*

Proof. "$\Rightarrow$": If $\mathcal{P}$ has no infeasible code, every statement $st \in \delta$ is part of some feasible path π_{st}. The set $\bigcup_{st \in \mathcal{F}(\delta)} \pi_{st}$ is a feasible path cover of $\mathcal{F}(\delta)$.

"$\Leftarrow$": suppose there is a feasible path cover of $\mathcal{F}(\delta)$. Let $st \in \delta$ be a statement. Since $\mathcal{F}(\delta)$ is maximal, there is a statement $st' \in \mathcal{F}(\delta)$ such that $st' \preceq st$. Since there is a feasible path cover of $\mathcal{F}(\delta)$, there is a feasible path that contains st'. Since $st' \preceq st$ this path also contains st. Thus every statement is contained in some feasible path and $\mathcal{P}$ has no infeasible code. $\square$

The theorem shows that the problem of detecting infeasible code can be understood as a path cover problem on an effectual set. By definition, the code of two equivalent statements $st \simeq st'$ is either for both infeasible or for neither. Thus, from knowing for each statement in an effectual set whether it is infeasible code, we can easily identify all minimal statements that are infeasible code. In the following we show that in reducible control flow graphs we can even identify all infeasible code. A control flow graph is reducible if removing all its back edges yields an acyclic graph. Recall that any path that contains a back edge has a loop that contains the back edge. We first show this for acyclic control flow graphs.

Lemma 1. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be an acyclic program. A statement $st \in \delta$ is infeasible code, if and only if every statement st' which is minimal with respect to $\preceq$ and for which $st' \preceq st$ holds is infeasible.*

Proof. If there is a feasible statement st' with $st' \preceq st$, there exists a feasible path which contains st' and therefore also contains st. Thus, st is feasible.

To show the other direction, we define a statement st to be *bad* if every minimal element st' with $st' \preceq st$ is infeasible but st itself is feasible. We need to show that there are no bad statements. For the sake of contradiction, suppose st is a bad statement that is minimal among all bad statements. Let st_1 be some minimal element with $st_1 \preceq st$. By our assumption st_1 is infeasible. Let π_1 be an infeasible path that contains st_1 and therefore necessarily also contains st, see Figure 2. W.l.o.g. we assume that when traversing π_1 from the source to the sink we first encounter st and then encounter st_1 (otherwise we invert the directions of all edges). Since st is feasible, there is a feasible path π_2 that contains st. Since π_2 does not contain st_1, after passing through st it must leave the path π_1 before reaching st_1. Let st_2 be the first edge on π_2 encountered after passing st that is not on π_1. We now show that $st_2 \preceq st$. Suppose this is not the case, then there is a path π_3 that contains st_2 but not st. We construct a path that contains st_1 but not st giving a contradiction: This path is obtained by starting along the path π_3

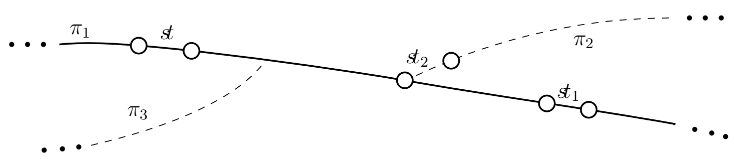


Fig. 2. The elements used in the proof of Lemma 1

up to the starting vertex of the edge st_2 and then following π_1 until the end. This path cannot exist, therefore we conclude $\mathit{st}_2 \preceq \mathit{st}$. We have $\mathit{st} \npreceq \mathit{st}_2$, since there is a path, namely π_1, that contains st but not st_2. Finally note that st_2 is bad since it is feasible and has the property that all statements st' with $\mathit{st}' \preceq \mathit{st}_2$ in particular fulfill $\mathit{st}' \preceq \mathit{st}$ and are thus infeasible. This is a contradiction to our minimal choice of a bad statement st and shows the theorem. □

For the curious reader, we remark that there is no equivalent theorem for the case of path-vertex covers (see [2] for an example). We now extend the lemma from acyclic graphs to reducible graphs.

Theorem 2. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program with a reducible control flow graph.*

1. *A statement $\mathit{st} \in \delta$ is infeasible code, if and only if every statement st' which is minimal with respect to $\preceq$ and for which $\mathit{st}' \preceq \mathit{st}$ holds is infeasible.*
2. *If a set of feasible paths covers all feasible statements within an effectual set $\mathcal{F}(\delta)$, then the set of paths covers all feasible statements.*

Proof. (Part 1). Given a reducible control flow graph $\mathcal{P}$ let $\mathcal{P}'$ be the graph obtained by redirecting all endpoints of back edges into the sink. Since $\mathcal{P}$ is reducible, $\mathcal{P}'$ is acyclic. Abusing terminology, for a back edge st in $\mathcal{P}$, we refer to the redirected back edge in $\mathcal{P}'$ also as st.

Claim: For statements $\mathit{st}, \mathit{st}'$ we have $\mathit{st}' \preceq \mathit{st}$ in $\mathcal{P}$ if and only if $\mathit{st}' \preceq \mathit{st}$ in $\mathcal{P}'$. To see the claim it suffices to observe that for every complete or infinite path π in one of the programs $\mathcal{P}$ or $\mathcal{P}'$ that uses a set of statements δ' there is a complete or infinite path π' in the other program that uses a (not necessarily strict) subset $\delta'' \subseteq \delta'$ of the statements. For a path π in $\mathcal{P}$ this is easy to see. We now show this for a path π in $\mathcal{P}'$. Let δ' be the set of statements on the path π. First note that π is finite since $\mathcal{P}'$ is acyclic. If the last edge of π does not correspond to a back edge in $\mathcal{P}$ then π is also a complete path in $\mathcal{P}$. Otherwise, if the last edge of π is a back edge in $\mathcal{P}$ then this edge closes a loop. By repeating this loop indefinitely, we obtain a path whose set of statements is the same as that of π. Either way, we obtain a path in $\mathcal{P}$ with the desired properties, showing the claim.

In $\mathcal{P}'$ we define a complete path to be feasible if its projection to $\mathcal{P}$ is a subgraph of some (complete) feasible path of $\mathcal{P}$. With this definition, a statement st is feasible in $\mathcal{P}$ if and only if it is feasible in $\mathcal{P}'$. Moreover, having now an acyclic control flow graph, Theorem 2 applies. Since feasibility of a statement and the "$\preceq$" relation are equivalent in $\mathcal{P}$ and $\mathcal{P}'$, Part 1 of the theorem follows.

(Part 2). To show Part 2 of the theorem let $\mathscr{s}$ be feasible code. Then, by the first part of the theorem, there is a minimal element $\mathscr{s}' \preceq \mathscr{s}$ that is feasible code. Any path cover that covers all feasible statements in an effectual set $\mathcal{F}(\delta)$ covers $\mathscr{s}'$. A path that contains $\mathscr{s}'$ also contains $\mathscr{s}$ and the theorem follows. □

The proof of the theorem also provides us with a method to compute the relation "$\preceq$": Indeed, to compute the relation $\preceq$ of a program $\mathcal{P}$ we first construct the acyclic program $\mathcal{P}'$ obtained by redirecting back edges into the sink. Since the relation $\preceq$ described in [12] then coincides with the relation defined in this paper, we can then apply the method described in [12] for the computation of $\preceq$.

To make use of the connection between infeasible code, effectual sets, and path covers, we first design an efficient path cover algorithm.

4 A Path Cover Algorithm

In this section we describe an algorithm that finds a feasible path cover for a given reducible control flow graph. The feasible paths are not given explicitly. We rather assume an oracle answers queries from which we can infer which edges of the graph are coverable by some feasible path. Intuitively, we need to optimize our query strategy towards a method that quickly dismisses large portions of the graph as coverable, allowing us to focus on edges that are uncoverable.

Abstractly we have the following model: We are given a control-flow graph $\mathcal{P}$ and repeatedly query for the non-existence of a feasible path with certain properties. We assume an oracle is available that provides us with an answer that either proves that no feasible path with the desired properties exists, or with a counterexample in form of a feasible path that possesses the required properties.

In more detail, the oracle answers the following type of query: For a specified set of nodes δ', and specified positive integers $\ell, k \in \mathbb{N}$ with $\ell \leq k$, does *no* feasible path exist that contains at least ℓ, and at most k elements of δ'? We assume this Constrained Path Infeasibility query is answered by a call to the function $CPI(\delta', \ell, k)$. In Sections 5 and 6 we explain why we use specifically queries of this type and how to realize an oracle that answers them.

Being able to query for a path that contains at least a certain number of edges from a specified subset allows us to adapt the greedy algorithm to our scenario. The standard greedy algorithm for the set cover problem repeatedly chooses a set that covers a maximal number of previously uncovered elements. A classic result by Johnson [15] shows this algorithm to be an $O(\log(n))$ approximation in terms of the number of sets used, and this is best possible, unless $P = NP$ [18].

Description of the algorithm. The path cover algorithm PCA (Algorithm 1) takes as input a reducible control flow graph and outputs a set of feasible paths that cover all feasible code. The algorithm starts by computing an effectual set $\mathcal{F}(\delta)$. It maintains a subset δ' of $\mathcal{F}(\delta)$ which at any point in time contains all elements of $\mathcal{F}(\delta)$ that cannot be covered by a feasible path. It repeatedly queries the oracle, and if returned a path, removes the statements on that path from δ'. The algorithm also maintains an integer k which is an upper bound on the number of statements in δ' that may lie simultaneously on a feasible path.

Algorithm 1. Path Cover Algorithm PCA

Input: $\mathcal{P} = (S, \delta, \Sigma)$: A reducible control flow graph.
Output: A set of feasible paths that cover all feasible code of $\mathcal{P}$.

```
     compute an effectual set $\mathcal{F}(\delta)$
     $k \leftarrow \mathrm{cov}(\mathcal{F}(\delta))$
     $\delta' \leftarrow \mathcal{F}(\delta)$
     while $\delta' \neq \{\}$ do
 5:    $k \leftarrow \min\{k, \mathrm{cov}(\delta')\}$
       query $CPI(\delta', \lceil k/2 \rceil, k)$
       if the query returned a path $\pi$ then
         let $E(\pi)$ be the statements on the path $\pi$
         $\delta' \leftarrow \delta' \setminus E(\pi)$
10:    else
         if $k = 1$ then
           return all paths that were reported by queries
         else
           $k \leftarrow \lfloor k/2 \rfloor$
15:      end if
       end if
     end while
     return all paths that were reported by queries
```

Theorem 3. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program with a reducible control flow graph that has a unique sink, a unique source, and an unknown set of feasible paths. Let $\mathcal{F}(\delta)$ be an effectual set. Algorithm 1 returns a set of feasible paths that covers all feasible code of $\mathcal{P}$. If K is the size of the smallest set of feasible paths that covers all coverable elements in $\mathcal{F}(\delta)$, then Algorithm 1 performs at most $O(K \cdot \log(\mathrm{cov}(\mathcal{F}(\delta))))$ queries.*

A proof of Theorem 3 is given in the extended version of this paper [2]. As mentioned previously, the set cover problem cannot be approximated with an approximation ratio of $o(log(n))$ unless $P = NP$ [18]. In our algorithm we made use of the parameter $\mathrm{cov}(\mathcal{F}(\delta))$ to get a finer analysis of the number of queries.

In general every set cover problem can be modeled as a path cover problem on a graph with unique sink and unique source: Indeed, by taking the transitive closure of a directed path, any subset of the edges of the original path can be chosen to be simultaneously on a feasible path. The inapproximability result thus applies to our path cover problem as well, and in this sense our algorithm is optimal with respect to the number of queries. However, control-flow graphs are not arbitrary graphs, and it might be possible to improve beyond the inapproximability ratio by using properties inherent to control-flow graphs.

5 Checking Constrained Path Infeasibility

In this section we explain how to construct the oracle $CPI(\delta', \ell, k)$ that checks for a program $\mathcal{P} = (S, \delta, \Sigma)$ whether there exists no feasible control-flow path π

that contains at least ℓ and at most k statements in $\delta' \subseteq \delta$. The call *CPI* returns such a feasible path π if it exists, otherwise it returns the empty path. We first devise a technique that allows us to modify any program so that this type of query can be answered. In particular with our modification, each query translates into a formula whose validity is equivalent to the non-existence of such a path. We use the concept of *reachability variables* introduced in [11] to monitor which statements are involved in an execution of a program $\mathcal{P}$.

So far, our approach is independent of a particular programming language. In the following, we require that our programming language is expressive enough to support variables with numeric types and assignment statements.

Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program and $\delta' \subseteq \delta$. For each statement $st \in \delta'$, we create an auxiliary *reachability variable* r_{st} in $\mathcal{P}$ which is initially zero. We replace a statement $st \in \delta'$ by the sequence $r_{st} := 1; st$. That is, every time st is executed r_{st} is assigned to one. Thus, after the execution of a path π in $\mathcal{P}$ the sum $\sum_{st \in \delta'} r_{st}$ is the total number of statements in δ' that occur on π.

Having introduced the reachability variables, the existence of a feasible path with at least ℓ and at most k statements from δ' in a program $\mathcal{P}$ can be checked using the weakest liberal precondition *wlp* of $\mathcal{P}$ and the postcondition $\neg(\ell \leq \sum_{st \in \delta'} r_{st} \leq k)$. This leads to the following theorem:

Theorem 4. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program, $\delta' \subseteq \delta$ a set and ℓ, k integers with $1 \leq \ell \leq k \leq |\delta'|$. The program $\mathcal{P}$ has no feasible path π, s.t. π contains at least ℓ and at most k statements from δ' if and only if the formula*

$$CPI(\delta', \ell, k) := wlp(\mathcal{P}, \neg(\ell \leq (\sum_{st \in \delta'} r_{st}) \leq k))$$

is universally valid in the program augmented with reachability variables.

A proof of Theorem 4 is given in the extended version of this paper [2]. In principle, we could design the query function *CPI* to work for arbitrarily complicated properties that are based on the reachability variables. In particular for any first order formula over the reachability variables we can obtain a theorem analogous to the one just presented. However, for actual implementations of the oracle, the complexity of the queries may alter the query response time, as we show in our experiments in Section 7. Our choice of query type is motivated by the fact that linear inequalities in practice can be handled well by theorem provers, and that this type of query suffices to construct an algorithm that is optimal with respect to the number of queries.

Up to this point neither the algorithm *PCA* nor the realization of *CPI* perform any abstraction. That is:

Lemma 2. *Given a sound, complete implementation of CPI, algorithm PCA is a sound and complete method to detect all infeasible code in a program.*

The proof of Lemma 2 follows directly from Theorem 3 and 4. However, an implementation of *CPI* needs to compute the weakest liberal precondition of a program, which is an undecidable problem in general. We will thus not be able to design an implementation of CPI which makes algorithm PCA simultaneously sound and complete. In the following, we describe a sound implementation of PCA which uses a sound *wlp* computation presented in [11].

6 Implementation

We now describe our implementation of a sound tool that detects infeasible code. Our implementation takes a Boogie program [16] as input, augments it with reachability variables, then applies the algorithm described in Section 4 and returns a subset of the infeasible statements of the program. We implement the constrained path infeasibility queries CPI using the sound over-approximation of the weakest liberal precondition presented in [11]. Soundness, in this case, means that for any path that has a feasible execution in the original program, there is a corresponding path with a feasible execution in the abstract program. If our translation were not sound, we might report infeasible paths that have feasible executions in the original program. We stress that this notion of soundness is dual to the notion of soundness used in verification.

Computing a formula representation of an over-approximation of the weakest (liberal) precondition of a program is a common technique [9, 10, 12, 14, 17]. It involves two steps: 1.) compute a loop-free abstraction of the input program, 2.) compute a formula representation of the weakest (liberal) precondition of the loop-free program.

Compute a loop-free abstraction. Given a Boogie program, we use the abstract loop unrolling presented in [12]. Loops are unrolled three times. The first unrolling represents the first iteration of the loop. The third unrolling represents the last iteration of the loop. Any other iteration is represented by the second, abstract unrolling. To retain soundness of the abstraction, non-deterministic assignments to all variables modified by the loop are added before and after the abstract unrolling. This abstraction is sound as it preserves the set of feasible executions of the original program. A proof of soundness is given in [12].

Compute weakest liberal precondition. For the loop-free program we perform a single-assignment transformation (e.g., [5]). We introduce an auxiliary variable for each assignment statement such that each variable is only written once. The resulting program is passive in a way that it does not change its state.

For a program $\mathcal{P}$, we denote the result of introducing reachability variables, eliminating loops, and altering the code to single assignment form by $trans(\mathcal{P})$. For $trans(\mathcal{P})$, we can compute a formula representing the weakest liberal precondition straightforwardly (see, e.g., [1, 11, 14, 17]).

In our implementation we can now use an automated theorem prover to check the satisfiability of the negation of the formula CPI from Theorem 4:

$$VC(\delta', \ell, k) := \quad \ell \leq \textstyle\sum_{st \in \delta'} (r'_{st}) \leq k \wedge wlp\,(trans(\mathcal{P}), false)\,,$$

where r'_{st} refers to the last incarnation introduced by the single assignment transformation. If the theorem prover is able to prove VC unsatisfiable for given

δ', ℓ, k we know that there exists no feasible path in the over-approximation of the program that contains at least ℓ and at most k statements in δ', and from the soundness of *trans* it follows that there is no feasible path in the original program either. Together with the algorithm *PCA*, this gives us a sound tool to detect infeasible code. We now compare this tool with existing techniques.

7 Evaluation

In this section we compare 3 algorithms to detect infeasible code in terms of theorem prover calls and computation time. We compare *PCA* presented in this paper, *Doomed* [11], which checks a minimal set of statements on loop-free programs, and *DoomedCE* [12], which performs the same checks as Doomed but utilizes the counterexamples emitted by the theorem prover to avoid redundant queries. Note, that DoomedCE can be considered a special case of PCA, where $\ell = 1$ and k is set to the number of statements.

We do not need to consider detection rate since all algorithms use the same sound weakest liberal precondition computation (which is part of the Boogie program verifier) and thus report the same infeasible code. For a comparison of detection rate with, e.g., Findbugs we refer to [12].

Experimental Material. To evaluate the performance of our algorithm, we use a set of 100 randomly generated Boogie procedures. We use generated programs because this allows us to vary the number of diamond shapes in the control-flow graph freely and no translation from a high-level language to Boogie that preserves the set of feasible executions is required. Existing translations from high-level languages into unstructured languages are not suitable for our algorithms since they over-approximate the set of infeasible executions to retain soundness w.r.t. partial correctness proofs. The threat to validity which arises from generated programs is discussed at the end of this section.

Each generated procedure has between 150 and 1500 lines of code and modifies up to 50 unbounded integer. The body of a procedure contains a sequence

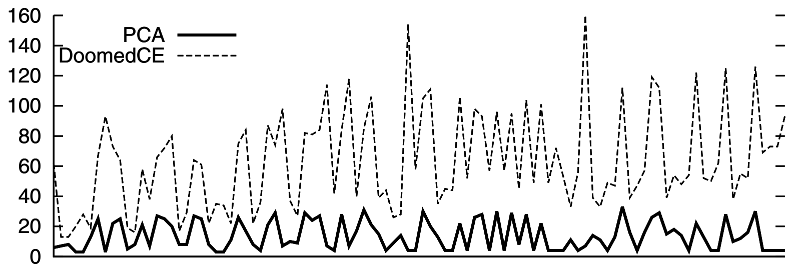


Fig. 3. Comparison of the number of queries. The x-axis ranges over the tested procedures, sorted by increasing length, the y-axis indicates the total number of theorem prover calls.

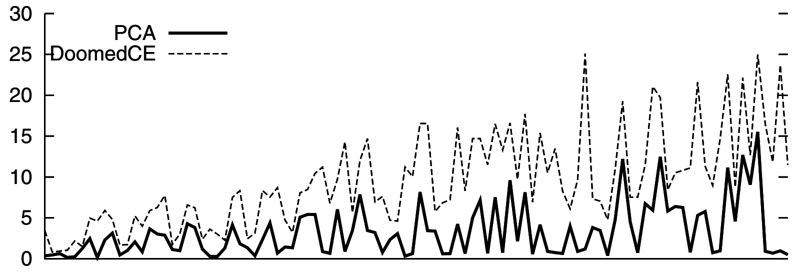


Fig. 4. Comparison of the computation time per program for PCA and DoomedCE. The x-axis ranges over the tested procedures, sorted by increasing length, the y-axis displays computation time in seconds.

of 5 to 10 conditional choices (diamond shapes) each with up to 5 nested conditional choices or loops. Besides the nested conditionals and loops, each block has between 3 and 5 statements. The statements are either assignments of expressions to variables, or assertions. All experiments are run several times on a standard laptop computer with ample memory.

Comparison of the algorithms. All algorithms identify 23997 out of 116925 statements to be infeasible code. *PCA* uses 1383 theorem prover calls and a total time of 324 sec to identify all infeasible code; Doomed uses 8942 theorem prover calls and 1309 seconds; DoomedCE uses 6365 queries theorem prover calls and 948 seconds. By its definition Doomed uses one query for each element in the effectual set. The algorithm DoomedCE covers the effectual set by querying 71% of the elements, and PCA is able to cover the set by querying 15% of the elements. In terms of computation time, DoomedCE needs 72% of the computation time of Doomed, and PCA needs 24% of the time used by Doomed.

Figure 3 compares the number of queries of PCA and DoomedCE in more detail. As expected, we can see that the number of queries for PCA is drastically lower than the number of queries for DoomedCE.

Figure 4 compares the computation time per procedure for algorithms PCA and DoomedCE. We can see that, even though PCA uses more expensive oracle queries than DoomedCE, there is a significant speedup due to the reduced number of queries.

Threats to Validity. The randomly generated programs is the main internal threat to validity. However, they allow us to control the shape of the control-flow graph and, in particular, the number of diamond shapes which is important to show the benefit of the path cover algorithm. The generated programs are of a very simple nature. They do not use complex types or a heap. For all algorithms, each query reasons about the set of paths in the program, these queries will become proportionally more expensive if reasoning about a single path becomes more expensive. Thus, we expect that our observations will still hold for real programs. However, in our future work, we have to evaluate if control-flow graphs of this complexity occur in practice.

Another internal threat to validity of our experiments arise form the used theorem prover. In our experiments we use only Z3. Other theorem provers may have a different efficiency for our kind of query (i.e., the linear inequalities). However, to avoid using a slow integer theory solver, we could alternatively encode the sum of reachability variables as a boolean formula.

External threats to validity arise from the needed translation from some high-level language to Boogie.

Discussion of the results. The drawback of DoomedCE compared to PCA is its inability to influence the counterexamples produced by the theorem prover. A counterexample may cover many statements that have already been covered by previously found counterexamples. Yet, forcing the theorem prover to provide more useful counterexamples comes at the price of longer query times. How high this price is, depends on the program structure. For example, if a program consists of sequential but independent parts, there are feasible paths that provide useful information in all parts simultaneously.

The effectiveness of searching useful counterexamples is also influenced by the properties we check. It is to be expected that there are properties for which the queries presented in this paper are significantly more expensive than, e.g., the queries used in DoomedCE. For these cases, algorithm PCA may require noticeably more computation time than DoomedCE. However, our experiments indicate that, at least for null pointer dereference and definite assignment analysis, the presented approach yields a significant performance improvement.

As mentioned earlier, algorithm DoomedCE can be considered as a special variant of PCA. Both algorithms are part of a family of algorithms obtained by varying the variables ℓ and k used to call CPI. This indicates that, even though the presented approach is query optimal, there is still room for optimization to achieve optimal computation times.

8 Conclusion

We have shown that the detection of infeasible code can be seen as a set cover problem and, more importantly, that covering *all* feasible statements in an effectual set determines all feasible statements in a program.

We presented an algorithm that detects all infeasible code in a program which uses an optimal number of queries. Using our implementation, for various applications of infeasible code detection, we have experimentally shown a significant decrease in the computation time when compared to existing methods.

For our future work we see two promising directions. We will incorporate the path cover algorithm directly in a theorem prover. In fact, the presented algorithm can be seen as a strategy to force a theorem prover to search for a particular counterexample. Thus, implementing this directly in a theorem prover may lead to performance improvements by allowing reuse of information more efficiently.

Another direction of future work is the development of different strategies to realize *CPI* based on different approximations of the weakest liberal precondition. A sound implementation of *CPI* that is only required to preserve all feasible executions of a program (in contrast to verification contexts, where all infeasible executions must be preserved) can use a coarser approximation and may result in a significant performance improvement while maintaining a reasonable detection rate. For some properties, a very coarse abstraction of *wlp* may still be sufficient to identify most infeasible code.

The main usefulness of the presented approach is that it detects some common types of errors. It works without user interaction and, in particular, without any false warnings. Perhaps the most intriguing aspect is that it has the potential to become fast enough to be applied while typing.

Acknowledgements. This work is supported by the project ARV funded by Macau Science and Technology Development Fund, by the National Research Fund, Luxembourg, and the Marie Curie Actions of the European Commission.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2005, pp. 82–87. ACM, New York (2005)
2. Bertolini, C., Schäf, M., Schweitzer, P.: Infeasible code detection. Technical Report 455, United Nations University, IIST (November 2011)
3. Bertolino, A.: Unconstrained edges and their application to branch analysis and testing of programs. Journal of Systems and Software 20, 125–133 (1993)
4. Bertolino, A., Marré, M.: Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Trans. Softw. Eng. 20, 885–899 (1994)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 451–490 (1991)
6. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Dijkstra, E.W.: A discipline of programming. Prentice-Hall, Englewood Cliffs (1976)
8. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci. 217, 5–21 (2008)
9. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
10. Grigore, R., Charles, J., Fairmichael, F., Kiniry, J.: Strongest postcondition of unstructured programs. In: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs, FTfJP 2009, pp. 6:1–6:7. ACM, New York (2009)
11. Hoenicke, J., Leino, K.R., Podelski, A., Schäf, M., Wies, T.: It's Doomed; We Can Prove It. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)

12. Hoenicke, J., Leino, K.R., Podelski, A., Schäf, M., Wies, T.: Doomed program points. Form. Methods Syst. Des. 37, 171–199 (2010)
13. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2004, pp. 132–136. ACM, New York (2004)
14. Janota, M., Grigore, R., Moskal, M.: Reachability analysis for annotated code. In: Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, SAVCBS 2007, pp. 23–30. ACM, New York (2007)
15. Johnson, D.S.: Approximation algorithms for combinatorial problems, vol. 9, pp. 256–278. Academic Press, Inc., Orlando (1974)
16. Leino, K., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
17. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. 93, 281–288 (2005)
18. Raz, R., Safra, S.: A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, STOC 1997, pp. 475–484. ACM, New York (1997)
19. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 245–256. IEEE Computer Society, Washington, DC, USA (2004)

Author Index