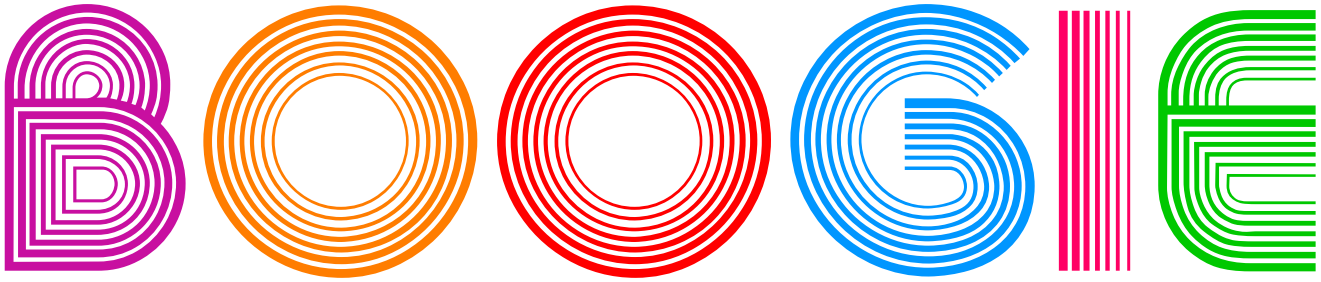

2nd International Workshop on Intermediate Verification languages

Proceedings

July 2012
Berkeley - California - USA

Preface

These informal proceedings contain the papers presented at BOOGIE 2012, the Second International Workshop on Intermediate Verification Languages, held on July 8, 2012 as part of the 24th International Conference on Computer Aided Verification (CAV) in Berkeley, California, USA.

Improving software correctness and reliability is of utmost importance, and therefore advancing the state-of-the-art in software verification is crucial. However, software verifiers are complex pieces of software, and the entry bar for building a software verifier is set very high. Requiring multiple man-years to develop a verifier impedes researchers from making fast advances in the area. Furthermore, having a large number of programming languages and available reasoning engines in use today is also a barrier as building a dedicated software verifier for each of them is not a viable solution. Therefore, similarly to compiler community in the past, software verification community started relying on intermediate verification languages (IVLs) to bridge the gap between front-end input languages and back-end reasoning engines. Implementing a verifier on top of an existing IVL infrastructure dramatically reduces entry development costs and effort. This workshop aims to bring together and foster collaboration among researchers and practitioners who work or rely on IVLs and the infrastructure behind them.

The proceedings contain 3 accepted papers and abstracts of an invited talk and a tutorial that were presented at the workshop. Every submitted paper was reviewed by 3 program committee members to ensure a high standard among the accepted papers.

I would like to thank the authors for their submissions, the program committee members for their reviewing efforts, the invited speakers for further enriching the workshop program, and the CAV conference organizers for accepting this workshop as a co-located event of the conference.

June 18, 2012
Salt Lake City, Utah, USA

Zvonimir Rakamarić

Program Committee

Dino Distefano	Queen Mary University of London
Jean-Christophe Filliâtre	CNRS
Michal Moskal	Microsoft Research
Peter Mller	ETH
Shaz Qadeer	Microsoft Research
Zvonimir Rakamarić	University of Utah
Stephen Siegel	University of Delaware
Ofer Strichman	Technion
Thomas Wies	New York University
Jochen Hoenicke	University of Freiburg

Table of Contents

Toward a Shared Infrastructure for Code Checking, Angelic Execution, Debugging, and Synthesis (Invited Talk)

Emina Torlak
University of California, Berkeley, USA
emina@alum.mit.edu

Decision procedures have helped automate key aspects of programming: coming up with a code fragment that implements a desired behavior (synthesis); establishing that an implementation satisfies a desired property (code checking); locating code fragments that cause an undesirable behavior (debugging); and running a partially implemented program to test its existing behaviors (angelic execution). Each of these aspects is supported, at least in part, by a family of formal tools. Most such tools are built on infrastructures that are tailored for a particular purpose, e.g., Boogie for verification and Sketch for synthesis. But so far, no single infrastructure provides a platform for automating the full spectrum of programming activities, making it hard to share advances (in encodings, abstractions, and domain-specific optimizations) across different families of tools.

This talk outlines a path toward a shared infrastructure for computer-aided programming, drawing lessons from a decade of collective experience in using relational constraint solvers to design and analyze software. We start with a relational view of the heap pioneered in the early work on the Alloy language and Analyzer. We then derive an encoding of programs in the bounded relational logic of Kodkod, which extends Alloy with support for partial models. Next, we show by example how to use such an encoding to build a program checker, an angelic oracle, an automated debugger, and a template-based synthesis tool. We conclude by discussing some of the challenges involved in lifting the low-level commonalities between these systems into an efficient infrastructure for prototyping, embedding, and synthesis of programming tools.

Combining Interactive and Automated Theorem Proving using Why3 (Invited Tutorial)

Jean-Christophe Filliâtre
CNRS Université Paris Sud
jean-christophe.filliatre@lri.fr

Why3 is a platform for deductive program verification. It features a rich logical language with polymorphism, algebraic data types, and inductive predicates. Why3 provides an extensive library of proof task transformations that can be chained to produce a suitable input for a large set of theorem provers, including SMT solvers, TPTP provers, as well as interactive proof assistants. In this tutorial, we show how this technology is used to combine interactive and automated theorem provers in, though not limited to, the context of program verification.

Prototyping Static Analysis Certification using Why3

Pierre-Emmanuel Cornilleau

INRIA Rennes – Bretagne Atlantique, France

firstname.lastname@inria.fr

We present a new methodology to reduce the Trusted Computing Base (TCB) of static analysers through result certification using Intermediate Verification Languages (IVL) and automated provers. Our approach uses results of static analysers as untrusted invariants to prove in the IVL the absence of run-time errors from the analysed program. The standard use of IVLs to prove programs written in another language requires a compiler targeting the IVL. To avoid inclusion of a compilation tool in the TCB, we use an interpreter programed in the IVL and an explicit representation of programs. To prove the correctness of the interpreter, programs and analyser's results are axiomatised, and verification conditions are generated for each program and proved using automated provers.

1 Introduction

Software security can be improved by many means. One is to use static program analysers to detect errors at compile-time, or even prove the absence of run-time errors. However, such *proofs* rely on the *soundness* of the analyser: there is no verifiable proof object, only the fact that the analyser did not report any error. Hence the analyser is included in the Trusted Computing Base [16] (TCB): the set of all components critical to the security of the application. But a precise and efficient static analyser is a complicated piece of software and may not be acceptable in the TCB, depending on the application.

One solution to reduce the TCB is to obtain a *foundational* proof that the program is free of run-time errors, as coined by Appel [1]: a proof relying only on a proof-checker and a formal semantic definition of run-time safety. It amounts to counting the proof-checker in the TCB in place of the analyser. A foundational proof of safety can be obtained by certifying the analyser inside a proof-checker. Pichardie *et. al* [9, 12] formalised the abstract interpretation framework [11] in the Coq proof-assistant and used it to prove the soundness of several program analysers. This approach requires to develop and prove in Coq the whole analyser which is a formidable effort of certification and raises efficiency concerns, Coq being a pure lambda-calculus language. Another way to obtain a foundational proof of safety is to certify, inside the proof-checker, a verifier of analysis result rather than the analyser. Besson *et. al* [5] applied this *result certification methodology* [18] to a polyhedral analysis, developing an analyser together with a dedicated checker whose soundness is proved inside Coq.

Previous work in the Intermediate Verification Language (IVL) literature is devoted to doing proofs on programs written in other languages, and gave birth to ambitious tool suites, such as Krakatoa [14] for proving Java programs using Why or Spec# [3] to prove C# using Boogie, to name a few. In these approaches, programs are compiled to the IVL, and the resulting program is proved using the Hoare logic and Weakest Precondition (WP) calculus available for the IVL. The encoding of source language features relies on typing and static analysis performed by the compiler. Therefore, the TCB of these approaches contains both the WP calculus of the IVL and the compiler, including all the analysis it performs.

The contribution of this paper is a new methodology to certify the results of static analysers, based on intermediate verification languages and automated provers, and its instantiation on numerical analysis

over an instruction-based unstructured language in Why3. We use static analysers as untrusted invariant generators, and traditional Hoare logic and Weakest Precondition (WP) calculus to prove the absence of run-time errors. If the result of the analysis is correct, it can be translated in invariants strong enough for Hoare logic to prove the program correct. If the analysis made a mistake, then the properties we used as invariants will not be strong enough or will not be invariants at all.

To follow the result certification methodology, we need the equivalent of an automatic verifier, hence automatic proofs for the verification condition raised by WP calculus. To minimise the TCB, we propose to use an explicit representation of the program, in its original syntax and to design an interpreter for this language, written in an Intermediate Verification Language (IVL), Why3 [6] in the experiments. A program is represented by functions describing its structure and the result of the analyser is abstracted by predicates on execution states of the interpreter. Using an axiomatised description of the relation between these predicates and functions, we prove the interpreter correct under verification condition.As a result, the TCB contains neither a compiler for the source language nor a language specific verification condition generator. Finally, to check the result of an analysis we only need to prove automatically that it satisfies the verification conditions. For that we rely on Automated Theorem Provers usable from Why3, such as Z3 [15], Alt-Ergo [10] or E [17].

The rest of the paper is organised as follows. The language, its semantic and an interpreter are presented in Section 2 together with the axiomatisation of programs and analysis results. Section 3 presents the experimentation done on two numerical analysis. Section 4 concludes with a discussion on automation, limits of the approach, and further work.

2 Language and generic interpreter

We illustrate our approach on a simple instruction-based unstructured language. Programs manipulate integers and arrays of integers, and are structured with procedures.

Syntax. The syntax of the language is presented below using Why3 syntax. Integers and Boolean expressions only manipulate integer variables, the only way to access the content of an array being through dedicated instructions. Our case studies only need sums of variables and division by two, a restriction due to the analysis considered.

```
type expr 'var =
   | Var 'var
   | Plus (expr 'var) (expr 'var)
   | Minus (expr 'var) (expr 'var)
   | Half (expr 'var)
   | Cst int
```

```
type test 'var =
   | Le (expr 'var) (expr 'var)
   | Neq (expr 'var) (expr 'var)
   | Eq (expr 'var) (expr 'var)
   | Lt (expr 'var) (expr 'var)
   | And (test 'var) (test 'var)
```

```
type stmt 'var 'proc 'pp =
   | Assign 'var (expr 'var)
   | JumpIfFalse (test 'var) 'pp
   | Call 'var 'proc (list ( expr 'var)) (list 'var)
   | Skip
   | GetArray 'var 'var (expr 'var)
   | SetArray  'var (expr 'var) (expr 'var)
   | Length  'var 'var
```

The generic types of expressions, tests and statements are parametrized with types for variables names (**var**), procedures names (**proc**) and program points (**pp**). We use a limited set of instructions. We

can assign a variable with an expression, make a conditional jump, and call a procedure. We also have instructions to obtain the content of an array, to assign a cell in an array, or to obtain the length of an array. For example, the statement "`Call x f [`e_1`;`e_2`] []`" stands for a call to procedure `f` over two integer arguments e_1 and e_2, and no array argument. The result of the procedure will be assigned to the variable `x`. We do not use a `Return` statements, a procedure signature indicated a final program point instead, and the name of the variable containing the result.

We have defined these instructions in a Why3 theory `Stmt`, leaving abstract the types of variables, program points and function names. Those types are instantiated while describing a specific program, and are not needed to detail the semantics of the language. Furthermore, we axiomatise the structure of any program with the functions described in Listing 1 in a separate theory `Domain`, which obtains the type **`stmt`** using the Why3 import mechanism. This `Domain` theory is imported in a Why3 module `Exec` which implements a generic interpreter using these abstract functions.

```
theory Domain
   use import Stmt

   (* Basic types *)
   type var
   type proc
   type pp

   (* Program instructions and control flow *)
   function get_stmt (p:pp) : stmt
   function next_pp (p:pp) : pp

   (* Procedure prototypes *)
   function proc_parameters (f:proc) : list var
   function proc_parameters_array (f:proc) : list var
   function proc_result (f:proc) : var

   (*Relations between procedures and program points *)
   function proc_beginning (f:proc) : pp
   function proc_end (f:proc) : pp
   function proc_context (p:pp) : proc

   ...
end
```

Listing 1: Axiomatisation of programs

The axiomatisation of programs first declares (but leaves abstract) the basic types of program identifiers. The manipulation of those types is left to each particular program description, but the verification conditions involve universal quantification over basic types (program points, variable and procedure names), and the domains of these types should be finite.

The structure of a program is described using functions `get_stmt`, which gives the instruction of each program point, and `next_pp`, which describes the control flow of the program. Procedures' prototypes are accessed using `proc_parameters`, `proc_parameters_array`, and `proc_result`, which for each procedure returns respectively the list of integer parameters names, the list of array parameters names and the name of the variable containing the result. Functions `proc_beginning` and `proc_end` indicate respectively the beginning and end of each procedure, and the last function, `proc_context`,

gives for each program point the procedure to which it belongs. Those last three functions are needed because program points can be referred to globally, and not only in a particular procedure.

Semantics. The semantics of expressions is given by a natural semantics presented in Figure 1a, and interprets the usual arithmetic operations and tests. For instructions, we use the operational semantics described in Figure 1b. It can be seen as a small-step semantics except for procedure calls, which involve a transitive closure of the semantic relation.

A state `s` is a record `{|st : store ; ars : arrays ; cpp : pp|}` containing a store `s.st` (map of var to int), a map of arrays `s.ars` (map of var to array) and the current program point `s.cpp`. An **`array`** is modelled by a recording of its length (field `len`) and of a map of indexes to elements (field `elts`). Array bound checks are enforced using side-conditions on the semantic rules, therefore it is not possible to do out-of-bound accesses: in that case the semantics blocks.

The semantics is presented using standard notation for access in records (`record.field`), access in maps (`map[key]`) and update in maps (`map[key<-value]`). Maps and records are functional, hence update in a map denotes a (new) map and records cannot be updated, only created using the Why3 notation (`{|`$field_1$ `=` $value_1$ `;` ... `;` $field_n$ `= value_n |}`). For example, in the interpretation of `Length x a`, the term `s.ars[a].len` refers to the field `len` of the array `a` in the state `s`. To assign this value to the variable `x`, we simply update the store `s.st`, and this new store denoted by `s.st[x<-s.ars[a].len]` is used to build the next state.

The functions used in the inference rules are referring to the axiomatisation of programs. They belong to a Why3 theory, not an actual implementation (module), hence cannot fail and do not need Hoare triples. Functions `mk_store` and `mk_arrays` are simple auxiliary functions which build, from a list of parameters and a list of arguments, maps from parameters to values (resp. a store and a mapping from variables to arrays). They do not involve any check on list length. If there is not enough arguments, the remaining parameters are not assigned, ans if there is too many arguments, they are discarded. The new store and array map are used to build the initial state *init* of a procedure call, which is reduced to a state *end*, hence the use of a transitive closure $init \longrightarrow^* end$. This state *end* holds the result of the procedure call. The semantic rules `GetArray` and `SetArray` make explicit the array bound check. Array contents are then safely manipulated as maps.

Interpreter. The natural semantics of expressions and tests is a total function from store and expressions to integers and Booleans. It is implemented in purely functional style in a Why3 theory, and used in formulae, under the names `evalexpr` and `evaltest`.

The operational semantics is very close to an actual interpreter, but is a blocking semantics, some states having no successors, and some programs may never terminate. For these reasons, the interpreter needs to be an implementation, defined in a Why3 module, with pre and postconditions. It is composed of two mutually recursive functions, `exec_stmt`, which given a state calculates the next step according to the semantics, and `exec`, which performs big-step reductions for procedure calls.

The function `exec_stmt` implements the operational semantics. Instead of accessing and updating arrays using their internal representation as maps, the interpreter uses two functions, `set_array` and `get_array`, which have as precondition the required check on indexes. This way, the constraints expressed in the semantics are turned, through the weakest precondition calculus of Why3, into verification conditions, which need to be discharged to prove the interpreter.

$$\text{Var}\frac{v \in Var}{(st, v) \Rightarrow st[v]} \qquad \text{Cst}\frac{n \in \mathbb{N}}{(st, n) \Rightarrow n} \qquad \text{Add}\frac{(st, e_1) \Rightarrow n_1 \qquad (st, e_2) \Rightarrow n_2}{(st, (\texttt{Plus}\ e_1\ e_2)) \Rightarrow n_1 + n_2} \qquad \ldots$$

$$\text{True}\frac{}{(st, true) \Rightarrow_b true} \qquad \text{False}\frac{}{(st, false) \Rightarrow_b false}$$

$$\text{Lt}\frac{(st, e_1) \Rightarrow n_1 \qquad (st, e_2) \Rightarrow n_2 \qquad n_1 < n_2}{(st, (\texttt{Lt}\ e_1\ e_2)) \Rightarrow_b true} \qquad \ldots$$

(a) Natural semantics of expressions and tests

$$\text{Assign}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{Assign}\ x\ e \qquad (s.st, e) \Rightarrow n}{s \longrightarrow \{|st = s.st[x \leftarrow n]; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{JumpIfFalseT}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{JumpIfFalse}\ t\ p' \qquad (s.st, t) \Rightarrow_b true}{s \longrightarrow \{|st = s.st; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{JumpIfFalseF}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{JumpIfFalse}\ t\ p' \qquad (s.st, t) \Rightarrow_b false}{s \longrightarrow \{|st = s.st; ars = s.ars; cpp = p'|\}}$$

$$\text{Skip}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{Skip}}{s \longrightarrow \{|st = s.st; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{Call}\frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = \texttt{Call}\ x\ f\ ints\ arrays \\ args_1 = \texttt{mk\_store}(s.st, \texttt{proc\_parameters}(f), ints) \\ args_2 = \texttt{mk\_arrays}(s.ars, \texttt{proc\_parameters\_array}(f), arrays) \\ init = \{|st = args_1; ars = args_2; cpp = \texttt{proc\_beginning}(f)|\} \\ init \longrightarrow^* end \qquad\qquad end.cpp = \texttt{proc\_end}(f)\end{array}}{s \longrightarrow \{|st = s.st[x \leftarrow end.st[\texttt{proc\_result}(f)]]; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{GetArray}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{GetArray}\ x\ a\ e \qquad (s.st, e) \Rightarrow n \qquad 0 \leq n < s.ars[a].len}{s \longrightarrow \{|st = s.st[x \leftarrow s.ars[a].elts[n]]; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{SetArray}\frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = \texttt{SetArray}\ a\ e_1\ e_2 \qquad (s.st, e_1) \Rightarrow i \qquad (s.st, e_2) \Rightarrow n \\ 0 \leq i < s.ars[a].len \qquad\qquad ar1 = s.ars[a].elts \\ ar2 = \{|len = s.ars[a].len; elts = ar1[i \leftarrow n]|\}\end{array}}{s \longrightarrow \{|st = s.st; ars = s.ars[a \leftarrow ar2]; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

$$\text{Length}\frac{\texttt{get\_stmt}(s.cpp) = \texttt{Length}\ x\ a}{s \longrightarrow \{|st = s.st[x \leftarrow s.ars[a].len]; ars = s.ars; cpp = \texttt{next\_pp}(s.cpp))|\}}$$

(b) Semantics of instructions

Figure 1: Semantics

```
let get_array (a:array) (i:int) =
{0 ≤ i < a.len}
       a.elts[i]
{result = a.elts[i]}
```

```
let set_array (a:array) (i:int) (v:int) =
{0 ≤ i < a.len}
    {|len = a.len; elts = a.elts[i<-v]|}
{result = {|len = a.len; elts = a.elts[i<-v]|}}
```

For these verification conditions to be provable, we need to give more information about the evolution of the state during an evaluation. The same way we axiomatised programs, we axiomatise analysis results: we declare some predicates standing for program invariants, presented in Listings 2 and 3. The predicates `pre` and `post` (resp. `proc_pre` and `proc_post`) encode pre and postconditions for each program point (resp. for each procedure). The other predicates are used to simplify the proof of the analysis results. The predicate `proc_inv_state` checks the invariance of the parameters of a procedure between two states, which makes it easier to state and check invariants establishing a relation between the arguments and the result of procedure calls; `successors` establishes a succession between program points, and `proc_inv_call_back` explicits the relation between procedures pre and postconditions.

```
predicate pre state pp
predicate post pp state
predicate proc_pre proc state
predicate proc_post proc state
```

Listing 2: Analysis results axiomatisation

```
predicate proc_inv_state proc state state
predicate proc_inv_call_back proc state state
predicate successors pp state
```

Listing 3: Supplementary predicates

The intended meaning of these predicates is made explicit in Why3 theorems, stating, for example, that for each program point, the postcondition is provable under the precondition. Listing 4 details the case of assignments. Remark that the semantics of statements appears in those theorems, here `sta2` is the result of an assignment; in the case of array accesses, theorems state that the precondition of such program points should be sufficient to prove that this access is possible. Listing 5 presents another crucial theorem, explaining that if a state `s` satisfies a postcondition and if it is not the end of a procedure, it must also satisfy the precondition of the next program point `s.cpp`. Other theorems deal in the same way with other statements, state that `proc_inv_state` is a reflexive and transitive relation or that two successive program points belong to the same procedure.

```
axiom pw_assign :
  ∀ st:store, ars:arrays, p:pp, v:var, e:expr .
    let sta1 = {|st = st ; ars = ars ; cpp = p|} in
    pre sta p  →
    get_stmt p = Assign v e  →
      let sta2 = {| st=st[v<- evalexpr st e] ; ars = ars ; cpp = next_pp p |} in
      proc_inv_state (proc_context p) sta1 sta2  ∧  post p sta2
```

Listing 4: Assignment correctness theorem

Once the interpreter is annotated according to Listing 6, with the predicates defined previously, provable verification conditions can be obtained, establishing the correctness of the interpreter. The axiomatisation of analysis results in several theorems enforces a clean splitting of the initial problem in several simpler ones with no growth of the TCB. To certify the result of a static analyser on a particular program, we only have to prove the axiomatisation theorems on an implementation of the axiomatising functions and predicates (Listing 1, 2 and 3). The generic proof of the interpreter ensures that these theorems are sufficient to prove the program free of run-time errors.

```
axiom functions_inv_exec_stmt_back :
  ∀ f:proc,  s:state, p:pp .
    not (p = resolve_end f)   →
    post p s  →
      pre s s.cpp
```

Listing 5: Point-wise invariance theorem

```
let rec exec_stmt  (context:proc)  (s:state)  : state =
  { pre s s.cpp  ∧   context = proc_context s.cpp }
      ...
  { successors s.cpp result  ∧ post s.cpp result
      ∧ context = proc_context result.cpp
      ∧ proc_inv_state (proc_context s.cpp) s result }

with exec (f:proc) (s0:state) (s:state) : state =
  { proc_pre f s0  ∧ proc_inv_state f s0 s  ∧ pre s s.cpp
      ∧ f = proc_context s.cpp }
      ...
  { proc_inv_call_back f s0 result  ∧ proc_inv_state f s0 result
      ∧ proc_post f result }
```

Listing 6: Interpreter's Hoare triple

Trusted Computing Base. The soundness of this approach relies on the adequate modeling of the language, and on the Weakest Precondition calculus (WP) of Why3. To prove a particular analysis result on a specific program, one has only to prove the theorems used to prove the interpreter, establishing point-wise correctness of the result of an analysis on a particular program. Hence the first part of the Trusted Computing Base (TCB) is the Why3 tool and its WP calculus.

Programs are not transformed to an Intermediate Verification Language (IVL), as their representation is explicit, rather, the semantics of the language is described in the IVL of Why3 and program invariants (or conditions) are described using the logic provided in the IVL. No transformation on top of the IVL is required, hence no compiler is added to the TCB.

Goals are discharged using Coq or automatic solvers. The TCB depends on the tools chosen to do so. The use of an interactive proof assistant is acceptable when doing the interpreter proof, as it needs to be done only once ; but in a certification scheme, the proof of goals specific to a program should be done automatically, to make sure that no knowledge of the analysis is required from users, and that we do not include the analysis in the TCB. On the other hand, Why3 provides no facility for checking automatic provers results for now, and the TCB includes these rather complex tools. The certification of automatic prover results is an active field of research [8, 2, 4, 7] and completes any program verification work using automatic tools, eventually eliminating the automatic prover from the TCB, in favour of proof-checkers developed in proof-assistants, such as Coq or Isabelle.

Our approach is thus a middle-ground between a compilation approach (*e.g.*, Krakatoa, Spec#), whose TCB includes a compiler performing transformations and typing, and full program proofs (*e.g.*, done in standard Why3 or Boogie), whose TCB only includes the WP calculus, but are restricted to programs written in the IVL. In both cases, automatic provers are widely used to discharge verification conditions, and must be added to the TCB.

3 Certification of numerical analysis

To describe the results of our approach, we instantiate the scheme on two analysis performed on different programs. The internals of the analysis will not be described, as only the results are needed in a certification scheme. To obtain the invariants we used a web demo provided by Bertrand Jeannet [13], in the *box* (for the intervals) and *convex polyhedra (polka)* settings. All development can be found at `http://www.irisa.fr/celtique/ext/chk-sa-boogie`.

Instantiation of a program. First, to complete the presentation of the approach we present the representation of a program calculating the factorial of an integer. The program is explicitly represented in a Why3 theory `Fact` implementing the abstract functions of Listing 1. The theory `Fact` implement the types **`var`**, **`pp`** and **`proc`**. It also imports the theory `Stmt` (see Section 2), to use the types **`expr`**, **`test`** and **`stmt`** in the implementation of the functions `get_stmt` and `next_pp` (presented in Figure 2) together with all the other functions of Listing1 (which are not represented here but can be inferred from the comments, together with a more traditional description of the control flow). Those definitions are used to instantiate the theory `Domain` (see Section 2) with the theory cloning mechanism of Why3. The `Domain` theory defines some functions and predicates used in the statements of the axiomatising theorems as well as the types **`state`** and **`store`**. Theory `Fact` can now instantiate the results of the analysis and define the predicates `pre` and `post`. The only thing left to check the result of an analyser is to prove the axiomatisation theorems.

```
type var = a | b | c | t
type pp = PP0 | ... | PP14
type proc = Fact | Mul

function get_stmt (p:pp) : stmt =
  match p with
    (* Fact a returns b *)
    | PP0  →  JumpIfFalse (Le (Var a) (Cst 1)) PP3
    | PP1  →  Assign b (Cst 1)
    | PP2  →  Skip
    | PP3  →  Assign c (Minus (Var a) (Cst 1))
    | PP4  →  Call c Fact2 [Var c]
    | PP5  →  Call b Mul [Var c;Var a]
    | PP6  →  Skip
    | PP7  →  Skip
    (* Mul a b returns c *)
    | PP8  →  Assign c (Cst 0)
    | PP9  →  Assign t (Var b)
    | PP10  →  JumpIfFalse (Lt (Cst 0) (Var t)) PP14
    | PP11  →  Assign c (Plus (Var c) (Var a))
    | PP12  →  Assign t (Minus (Var t) (Cst 1))
    | PP13  →  JumpIfFalse (Lt (Cst 0) (Var t)) PP14
    | PP14  →  Skip
  end
```

Listing 7: Statements

```
function next_pp (p:pp) : pp =
  match p with
    (* Fact from PP0 to PP7*)
    | PP0  →  PP1      (* If *)
    | PP1  →  PP2
    | PP2  →  PP7
    | PP3  →  PP4    (* Else *)
    | PP4  →  PP5
    | PP5  →  PP6
    | PP6  →  PP7   (* EndIf *)
    | PP7  →  PP7
    (* Mul from PP8 to PP14*)
    | PP8  →  PP9
    | PP9  →  PP10
    | PP10  →  PP11 (* While*)
    | PP11  →  PP12
    | PP12  →  PP13
    | PP13  →  PP11 (* Done *)
    | PP14  →  PP14
  end
```

Listing 8: Control flow

Figure 2: Factorial

The expressions are given as terms, and operators are constructors of the algebraic types **`expr`** or

test. The `next_pp` function differs from the `successor` predicate on `JumpIfFalse` instructions. The former giving only the following program point in case the test evaluate to `true`, while the latter connect a program point to all its successors, whatever the results of the evaluation. In this aspect, `next_pp` only describes a sub-graph of the control flow graph. In this example the program points are numbered, but this numbering has no semantic value.

Instantiation on an interval analysis. The analysis calculates for each integer variable, at each program point, an interval to which belongs the value of the variable. An interval is a pair of bounds in $\mathbb{N} \cup \{+\infty, -\infty\}$. The abstraction of a state if thus a list of pairs (var * interval). For example, the abstract state [(a,$[-\infty, 0]$) ; (b,[3,5])] would be represented by the predicate `interval_1` .

```
predicate interval_1 (st:store) =
      let content_a = st[a] in
      let content_b = st[b] in
      content_a ≤ 0 ∧ 3 ≤ content_b ∧ content_b ≤ 5
```

The interval analysis only deals with integer variables, hence with the store of the current state. Why3 predicates, such as `interval_1`, will be used to implement all predicates from Listing 2 and 3, among which pre and postconditions (respectively `pre` and `post` predicates). The variables `content_a` and `content_b` are typed as standard Why3 `integers`, given that the type **`store`** is a shortcut for `map` **`var int`**. The constants `a` and `b` are elements of type **`var`**. The implementation of `pre` presented below, will simply apply the predicate `interval_1` or similar ones describing other intervals, on `s`, depending on the value of `p`.

```
predicate pre (s:state) (p:pp) =
   match p with
      | PP0 → interval_1 s.st
      ...
   end
```

Results. We ran the interval analysis on the factorial program described in Figure 2. This program does not manipulate any array, therefore could not make any run-time error, but it has procedure calls, and we wanted to first evaluate the efficiency of ATPs on the axiomatisation. For this experiment, we use Z3 (version 2.2) and Alt-Ergo (version 0.94) on a 3 years old laptop using 4 GiB of memory and a 2.93 GHz Intel Core 2 Duo under Linux. We used the Why3 backend *as shipped* to launch the solvers, and in particular did not try to find the most efficient settings for each solver. We present the results on only 2 different provers because they were sufficient for the experiment, and the different measures are given to illustrate the combination of different tools rather than to compare two particular tools.

All the axiomatisation theorems are proved by a combination of Z3 and Alt-Ergo. Z3 can prove alone all but 1 (out of 20) theorems in 0.2 seconds on average, whereas Alt-Ergo proves 8 in 0.9 seconds on average (but with a much higher variability), returns *unknown* in less than a second for 4, and reaches timeout (30s) for the others. One theorem is solved by neither of them. To prove it, we have two solutions: we can apply Why3 transformations to inline and split the goal, and then Alt-Ergo proves it in 2 seconds whereas Z3 still reaches timeout; or try to prove instantiations of the theorem on relevant program points and add them as lemmas. The instantiated lemmas are proved in 0.2 seconds by Alt-Ergo, but Z3 reaches timeout, and the theorem is now proved by Z3 in 0.3 seconds but Alt-Ergo still reaches timeout. Both ways to solve the problem can be applied automatically, as instantiations of the theorems can be stated in advance, and Why3 transformations applied a finite number of times.

Overall, theorems are discharged very quickly. However, the two ATPs we used exhibit very different behaviours. These differences were crucial for the proof of one of the theorems, which needs to be split in two different parts, each part being proved by different ATPs.

Instantiation on a relational analysis. The relational analysis chosen for this experimentation is a polyhedral analysis that gives linear relations between variables at each program point. As parameters of procedures are not reassigned, we also get, for each procedure call, relations between the arguments and the result. To illustrate the elimination of explicit array bound checks, this analysis not only deals with integer variables, but takes also into account the length of the arrays.

The abstraction of a state is a polyhedron, described by a set of linear inequalities on the variables and the length of arrays, as illustrated by the predicate `polyhedron_3` listed below. It gives constraints between the content of variables `a`, `b`, `s`, and the length of the array `t`. It differs from the predicate `interval_1` of the previous example as it needs the whole current state and not only the store, but will be used in the same way to implement the axiomatisation predicates.

```
predicate polyhedron_3 (s:state) =
  let length_t = s.ars[t].len in
  let content_a = s.st[a] in
  let content_b = s.st[b] in
  let stop = s.st[s] in
    content_a=0 ∧ content_b ≥ 0 ∧ - content_b + length_t -1  = 0 ∧ stop = 0
```

Results. The standard binary search program makes an interesting case study for a relational analysis, as it involves multiples array access at indexes between bounds indirectly related to the array length. As stated in Section 2, array bound checks are enforced in the interpreter through the use of array access functions, and appear in the corresponding axiomatisation theorems. If the analysis is precise enough, discovered invariants should be strong enough to prove the safety of array accesses.

In the binary search program an integer variable `s` is used as a surrogate Boolean variable, with value `1` and `0` standing for `true` and `false`, not intended to be related to the other variables. Nevertheless, the analyser used for this experiment discovered hidden but unnecessary relations between all the integer variables, and invariants tended to be very large, with lots of noise. This is not unusual but rather inevitable for static analysis and a certification scheme should be robust regarding such convoluted invariants. The interesting point is that those invariants were checked and were precise enough to ensure array access safety. Even if the analysis results were far from invariants used in manual program proof, no post-processing was necessary to prove the axiomatisation theorems.

Looking at the time taken by ATPs, we have a very clear distinction between the twenty theorems. Half of them are devoted to instructions, like the theorem from Listing 4. On most of them it is necessary to apply Why3 transformation, and one of them needs to be instantiated. This allows Alt-Ergo to prove them in 20 seconds on average, whereas Z3 still reaches timeout on all theorems except the instantiated lemma. Strangely, on this lemma Z3 outperforms Alt-Ergo, taking 1.5 seconds, a tenth of the time taken by Alt-Ergo. The other half is due to redundancies in the predicates representing static analysers results. This redundancies helped the ATPs to prove some of the theorems. On nine of them, Z3 took less than a second, whereas Alt-Ergo reaches timeout for one, returns unknown on 3, proves one of them in 40 seconds and the remaining 5 in less than 0.1 seconds. On the contrary, on the last one Z3 reaches timeout and Alt-Ergo takes around 0.1 seconds.

Overall, the two different provers behaved very differently, and there is almost no overlap between the theorems proved by one and those proved by the other. One takes less than a second or reaches timeout

(60 seconds), the other exhibiting a larger spectre of behaviours. They do not take as input the same standards, and Why3 does not use the same encoding for its high level features, therefore it is difficult to determine the reasons behind these differences. In any event, the experiments have demonstrated that the possibility to use different ATPs was invaluable to prove the axiomatisation theorems.

4 Conclusion and further work

Automation. The implementation of the *pre* and *post* predicates for the analysis chosen for the experimentation does not involve any quantification and can be easily stated with the *match with* construct of the Why3 language. Nevertheless, what seems to be simple proofs can be very hard for an automatic tool, as demonstrated our experiments. On the other hand, it is a simple case analysis, and Why3 provides simple transformations, namely *inlining* and *splitting*. These transformations result in equivalent set of goals, which can be discharged automatically by Alt-Ergo or Z3. This seems to indicate that two levels of automation may be needed to extend the approach to more complicated semantics: a very simple high level reasoning to eliminate constructs from the IVL and then efficient ATPs.

Another way to help the ATPs, is to instantiate the theorems on a particular program point. During our experiments, some of the full theorems could not be proved automatically, whereas instantiation of those theorems on relevant program points could be. And with these instantiated lemmas in their context, the ATPs were finally able to discharge the full theorem. These instantiations can be generated automatically, and no manual proof or prior knowledge is needed to obtain a full proof of the program. Some of the predicates used in the axiomatisation of analysis results could be expressed using combinations of `pre`, `post` and the functions describing the program, reducing the number of theorems needed to prove the interpreter. Nevertheless, introducing redundancy and intermediate predicate makes it easier for automatic provers to discharge the verification conditions, and the axioms introduced by these redundant predicates are discharged easily by ATP and do not introduce any noticeable overhead, as demonstrated by the experiment on the relational analysis.

Limits. If the soundness of the approach is formally proved, its completeness is not clearly established, as the axiomatisation theorems may not be appropriate for all analysers. They are sufficient, as the proof of the interpreter established, but they may not be necessary. The use of weaker theorems may complicate the proof of the interpreter, which is not a problem as it is done only once and may even simplify the automation of the proof of analysers results. However, it is not necessary in our case studies, as every verification condition is discharged using ATPs. Nevertheless the amount of automation is a concern. The statement of the semantics of the language and the axiomatisation theorems were carefully crafted, as minor syntactic differences can make a huge difference when discharging the verification conditions.

Summary and further work. This paper presents a static analyser results certification scheme minimising the Trusted Computing Base (TCB). To eliminate any compilation process from the TCB, programs are represented explicitly in a intermediate verification language. A generic interpreter is proved under an axiomatisation of programs and analysers results on programs. To prove a program free of runtime errors, we use ATPs to check that the analyser results on this program satisfy the axiomatisation.

Further work includes certification of non-numerical analysis and interpretation of richer languages. Current work can be systematically extended to non-numerical analysis, using the abstract interpretation framework, but numerical analysis were a favourable case, as the invariants were in a decidable fragment. To discharge the verification conditions raised by analysis built on semi-decidable domains

using automatic theorems provers remains an open issue. The extension of programs axiomatisation to richer languages, with more complicated memory model for example, may lead to complicated verification conditions as well. Finding a systematic approach or appropriate subset of the language to obtain axiomatisation theorems provable by ATPs is another open issue.

References

[1] Andrew W. Appel (2001): *Foundational Proof-Carrying Code*. In: *LICS 2001*, IEEE Computer Society, pp. 247–256.

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *CPP 2011*, *LNCS* 7086, Springer, pp. 135–150.

[3] Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2004): *The Spec# Programming System: An Overview*. In: *CASSIS 2004*, *LNCS* 3362, Springer, pp. 49–69.

[4] Frédéric Besson, Pierre-Emmanuel Cornilleau & David Pichardie (2011): *Modular SMT Proofs for Fast Reflexive Checking Inside Coq*. In: *CPP 2011*, *LNCS* 7086, Springer, pp. 151–166.

[5] Frédéric Besson, Thomas P. Jensen, David Pichardie & Tiphaine Turpin (2010): *Certified Result Checking for Polyhedral Analysis of Bytecode Programs*. In: *TGC 2010*, *LNCS* 6084, Springer, pp. 253–267.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2011): *Why3: Shepherd Your Herd of Provers*. In: *Boogie 2011*, pp. 53–64. Available at `http://proval.lri.fr/submissions/boogie11.pdf`.

[7] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell & Tjark Weber (2011): *Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL*. In: *CPP 2011*, *LNCS* 7086, Springer, pp. 183–198.

[8] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In: *ITP 2010*, *LNCS* 6172, Springer, pp. 179–194.

[9] David Cachera, Thomas P. Jensen, David Pichardie & Vlad Rusu (2005): *Extracting a data flow analyser in constructive logic*. *Theoretical Computer Science* 342(1), pp. 56–78.

[10] Sylvain Conchon, Evelyne Contejean & Johannes Kanig (2006): *Ergo : a theorem prover for polymorphic first-order logic modulo theories*. Available at `http://ergo.lri.fr/papers/ergo.ps`.

[11] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *POPL 1977*, ACM, pp. 238–252.

[12] Pichardie David (2005): *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. Ph.D. thesis, Université Rennes 1. In french.

[13] Bertrand Jeannet: *The Interproc Analyzer*. Available at `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`.

[14] C. Marché, C. Paulin-Mohring & X. Urbain (2004): *The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML*. *Journal of Logic and Algebraic Programming* 58(1–2), pp. 89–106.

[15] L. M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340.

[16] John M. Rushby (1981): *Design and Verification of Secure Systems*. In: *SOSP 1981*, ACM, pp. 12–21.

[17] Stephan Schulz (2002): *E – a brainiac theorem prover*. *AI Communication* 15(2-3), pp. 111–126.

[18] Hal Wasserman & Manuel Blum (1997): *Software reliability via run-time result-checking*. *Journal of the ACM* 44(6), pp. 826–849.

KIL: An Abstract Intermediate Language for Symbolic Execution and Test Generation of C++ Programs

Guodong Li
Fujitsu Labs of America, CA
gli@us.fujitsu.com

Indradeep Ghosh
Fujitsu Labs of America, CA
ighosh@us.fujitsu.com

Sreeranga P. Rajan
Fujitsu Labs of America, CA
sree.rajan@us.fujitsu.com

We present a declarative intermediate language KIL in a symbolic executor for C++ programs and show how to use KIL to control symbolic execution. KIL is an abstract language defined over LLVM bytecode; it provides a higher level model of C++ object operations and functions. KIL enables lazy function evaluation, object-level execution and reasoning, defining built-in efficient solvers, function summary based execution, and so on. Specifically, the symbolic executor first performs KIL level execution to build abstract path constraints, then uses these abstract constraints to guide bytecode level execution and produce concrete test cases. This method is able to avoid exploring many useless or duplicate paths. Experimental results show that KIL can help solve path constraints and produce test cases in much less time without losing source coverage.

1 Introduction

With the ubiquitous presence of software programs permeating almost all aspects of daily life, it becomes a necessity to provide robust and reliable software. Traditionally, software quality has been assured through manual testing which is tedious, difficult, and often gives poor coverage of the source code especially when availing of random testing approaches. This has led to much recent work in the formal validation arena. One such formal technique is symbolic execution [4, 1, 12, 8, 14] which can be used to automatically generate test inputs with high structural coverage for the program under test.

These widely used symbolic execution engines [4, 1, 12] currently are able to handle C or Java programs. We have reported a symbolic execution tool KLOVER [8] designed specifically for the automatic validation and test generation for C++ programs. Currently C++ is the language of choice for most low-level scientific and performance critical applications in academia and industry. Similar to [4, 1, 14], KLOVER works on the bytecode (in particular, LLVM bytecode [11, 6]) generated by a compiler.

While working in the bytecode level facilitates greatly the implementation of a symbolic executor (*e.g.* the tedious task of C++ syntax parsing is gone), many high-level details presented in the source language are lost during the translation to bytecode, making the executor difficult to perform higher level (*e.g.* abstract or modular) reasoning about the source programs. For instance, since the logical relations between the member functions of a C++ library class are blurred after the compilation, it is not easy for a symbolic executor to utilize such relations to expedite the execution and test generation procedure. After all, the symbolic executor sees only a set of unstructured bytecode instructions. As a consequence, the executor may easily get stuck in exploring unnecessary paths, and fail to produce valid test cases in a reasonable time. Furthermore, modern symbolic executors often use specific high-performance solvers to speed up the solving of specific constraints involving commonly-used data structures. For example, there is much work [2, 16, 5] on implementing string solvers and incorporating them into symbolic executors. This is not an easy task, and remains as an active research topic. A natural question to ask is whether we can make such effort easier and not ad-hoc.

To address these problems, this paper proposes using a declarative intermediate language KIL (named after KLOVER IL) to

- depict the (abstract) relations of the member functions of C++ classes and objects;
- use these relations to guide symbolic execution to avoid exploring useless paths in early phases;
- use function abstraction/summary and lazy function evaluation to search the state space better;
- enable early termination to avoid exploring duplicate paths and states;
- enable defining built-in efficient solvers directly in the source code;
- and more importantly, perform all these tasks in a unified, customizable and extensible framework.

With these optimizations, KLOVER is able to reduce the symbolic execution time by orders of magnitudes, yet produce the same set of valid test cases as before without losing any source code coverage! As far as we know, this work is the first effort to to apply abstract symbolic execution on C++ programs using a declarative intermediate language.

Our main contributions include introducing a rich constraint language in KLOVER and using this language to control symbolic execution and constraint solving. Although this paper focuses on C++ strings, our method can be used in other symbolic executors (either for test generation or bounded symbolic verification).

We organize the paper by first giving a motivating example, then describing KIL, and then showing how to use KIL to implement above-mentioned optimizations. We will also show some experimental results indicating the advantages of using KIL.

2 Backgroup and Motivation

Symbolic Execution. Symbolic execution performs the execution of a program on symbolic (open) inputs. It computes the effect of these symbolic inputs in the program using symbolic expressions. It characterizes each program path it explores with a path condition encoded as a conjunction of Boolean clauses. A path condition denotes a series of branching decisions. When the execution is finished, multiple path conditions may be generated, each corresponding to a feasible execution path of the code with respect to the symbolic inputs. The solutions to path conditions are the test inputs that will assure that the program under test runs along a particular concrete path during concrete execution. Typically a decision procedure such as an SMT (Satisfiability Modulo Theory) solver [13] is used to find the solutions and prune out false paths. Exhaustive testing is achieved by exploring all true paths.

KLOVER. KLOVER [8] is built on top of a symbolic execution engine KLEE [4] to handle C++ programs. It extends KLEE to handle virtually all C++ features, and uses various optimizations to make the tool efficient and scalable. KLOVER has successfully [1] unit-tested multiple industrial applications by Fujitsu including one with over 200k lines of code. Besides, our other symbolic execution tool GKLEE [9] addresses GPU (Graphical Processing Unit) programs and shares some components with KLOVER.

The C++ standard includes a library for all commonly used data structures and algorithms. Instead of using the standard implementation provided with GCC, we optimized the uClibc++ library [15] so as to improve the performance of symbolic execution. We compile this library into LLVM bytecode and load it into the executor dynamically. We will describe KLOVER's previous optimizations on C/C++ libraries and compare them with what we propose in this paper.

[1] *e.g.*, the drivers are constructed automatically, and the symbolic executor achieves $> 90\%$ source line coverage and $> 60\%$ branch coverage.

```
bool IsEasyChairQuery(string str) {
1:  // (1) check that str contains "/" followed by anything
2:  // not containing "/" and containing "EasyChair"
3:  int lastSlash = str.find_last_of('/');
4:  if (lastSlash == string::npos)
5:  { printf("exit (path) 1 \n"); return false; }
6:
7:  string rest = str.substr(lastSlash + 1);
8:  if (rest.find("EasyChair") == -1)
9:  { printf("exit (path) 2 \n"); return false; }
10:
11: // (2) Check that str starts with "http://"
12: // the compare function returns 0 if the compared sequences are equal
13: if (str.compare(0, 7, "http://"))
14: {  printf("exit (path) 3 \n"); return false; }
15:
16: // (3) Take the string between "http://" and the last "/".
17: // if it starts with "www." strip the "www." off
18: string t = str.substr(7, lastSlash-7);
19: if (!t.compare(0,4,"www.")) { t = t.substr(4); // imme. path 2}
20: // imme. path 1
21: // (4) Check that after stripping we have either "live.com" or "google.com"
22: if (!(t == "live.com") && !(t == "google.com"))
23: { printf("****** exit (path) 4 ****** \n"); return false; }
24:
25: // s survived all checks
26: printf("****** exit (path) 5 ****** \n");
27: return true;
}
```

Figure 1: A C++ version of the example program in [2].

2.1 Motivating Example

Figure 1 shows a C++ program manipulating strings. It is ported from the main benchmark program used in [2]. In order not to miss some valid paths, we must set the length of the input string str to be at least 29. Unfortunately, when using the original uClibc++ library implementation KLOVER fails to give a valid test case for the following path (the last constraint may lead to two paths in some executors) in 2 hours on a standard laptop. The handling of other paths encounter the same problem, although those with simpler path conditions need less time to process.

```
lastSlash != string::npos ∧
rest.find("EasyChair") != -1 ∧
str.compare(0, 7, "http://") != 0 ∧
!t.compare(0,4,"www.") ∧
t == "live.com" ∨ t == "google.com"
```

One main problem of using the default library implementation is that many new states will be spawned in the very beginning, while these states may lead to only invalid paths (and test cases) in subsequent execution. For example, consider the path that passes the checks !t.compare(0,4,"www.") at line 19 and t == "live.com" at line 22, the shortest valid input string (note the string's length is not fixed and can be any value from 0 to max_len (*e.g.* 255)) is "http://www.live.com/EasyChair" of length 29. All the trials on the cases with smaller lengths are fruitless. In fact, for these lengths, there is no need

for the executor to go into the bodies of the involved string APIs (including find_last_of, substr, find, compare, and $==) at all!

To see why executing the body of an API too early is bad, let's look at the following implementation of the compare function in uClibc++ [2]. The executor may spawn $O(2^n)$ new states /paths when executing this code, where n is the string's length; the situation becomes worse when the length is not fixed since we will have to iterate over all possible lengths to identify the minimum valid length.

```
int compare(const string& str) const {
  size_type rlen = length();
  if (rlen > str.length()) rlen = str.length();
  for (size_type i = 0; i < rlen; i++) {
    if (operator[](i) > str[i]) return 1;
    else if (operator[](i) < str[i]) return -1;
  }
  if (length() < str.length()) return -1;
  else if (length() > str.length()) return 1;
  return 0;
}
```

In [8] KLOVER proposes a method to avoid early state spawning by eliminating the branches in the function body. For a specific length, the following code produces only one path regardless of the values of the two input strings.

```
int compare(const string& str) const {
  size_type rlen = length();
  if (rlen > str.length()) rlen = str.length();
  int v = 0; // 1, 0 and -1 stand for gt, eq and lt respectively
  for (size_type i = 0; i < rlen; i++)
    v += (~(!v)+1) & ((operator[](i)>str[i]) - (operator[](i)<str[i]));
  v += (~(!v)+1) & ((length() > str.length()) -
                    (length() < str.length()));
  return v;
}
```

However, this method rapidly builds up long and complicated expressions and puts much heavier burden on SMT solvers, *e.g.*, the resulting expression v contains $n+1$ bit-wise operations. Moreover, it still suffers from the problem of exploring useless code, *e.g.*, it tries length values ranging from 0 to 28.

In [8] KLOVER proposes using an off-the-shelf string solver to handle the path constraints: during the execution, all the string operations are intercepted to build path conditions containing string operations. For such a path condition, the solver first extracts a set of constraints restricting the lengths and then solves them to obtain a (minimal) value for each length. After that, the solver fixes the length of each string, models each symbolic string with a bounded bit-vector, and solves the original constraint using SMT solvers (or automaton-based solvers). In principle, this method is similar to those in [2, 16, 5] but works on C++ strings. Since it avoids many useless paths caused by unoptimized API implementation, it can dramatically reduce the execution and solving time.

This leads to some interesting questions:

- Can we substantially reduce the burden of developing such solvers and embedding them into a symbolic executor working on bytecode?
- Can we reuse all the optimizations already in an efficient executor like KLEE when implementing these solvers?

[2] In this paper we will use simplified pseudo code to skip unimportant details.

τ	:=	$\mathtt{i1}, \mathtt{i2}, \ldots$	bitvector type
e	:=	$c : \tau$	constant
	|	$id : \tau$	variable
	|	$id[e]$	array read
	|	$id[e \mapsto e]$	array update
	|	$\mathtt{op_u}\ e$	unary operation
	|	$e\ \mathtt{op_b}\ e$	binary operation
	|	$\mathtt{op_t}(e,e,e)$	ternary operation

τ'	:=	τ	primitive type
	|	$\tau \times \cdots \times \tau$	vector type
τ_f	:=	$\tau \to \tau'$	function type
τ_p	:=	$\tau \to \mathtt{bool}$	predicate type
e	:=	$\lambda\, v.\, e(v)$	lambda expression
	|	$\forall v \in [e,e] : (e : \tau_p)\ v$	forall expression
	|	$\exists v \in [e,e] : (e : \tau_p)\ v$	exists expression
	|	$(id : \tau_f)(e, \ldots, e)$	function application
	|	$(id : \tau_p)(e, \ldots, e)$	predicate
	|	$\mathtt{f_{abs}}(e, \tau')$	uninterpreted abstraction

Figure 2: Main syntax of KIL

- • Can we customize how the executor explores the state space based on some abstraction knowledge such as the length constraints for strings?
- • Can such an approach be easily extended to handle other data structures rather than strings?

This paper gives positive answers to these questions by introducing an abstract immediate language KIL to control how the state space is explored and how constraint solving is performed with respect to the semantics of C++ functions. For instance, it is convenient to define in KIL a built-in string solver whose performance can rival with off-the-shelf string solvers.

3 The KIL Intermediate Language

3.1 Syntax and Evaluation Order

KIL extends KLEE's expression language by adding advanced expressions such as quantified expressions, function applications, object abstractions, *etc.*, plus mechansims to evaluate these advanced expressions in the symbolic engine. In Figure 2, we show the syntax of the primitive expressions inherited from KLEE (left) and the main advanced expressions introduced by KIL (right). A forall expression $\forall v \in [e_1, e_2] : p\ v$ indicates: when variable v's value is within the range $[e_1, e_2]$, predicate p holds for v. For example, $\forall v \in [1, 10] : f(v) \neq 100$ specifices that $f(v)$ is not equal to 100 when $1 \leq v \leq 10$. KIL also supports a more general form $\forall v\ \mathtt{with}\ p\ v : q\ v$ for predicates p and q. The formats for function application and predicate expressions are self-explanatory. The abstraction operation $\mathtt{f_{abs}}(e, \tau')$ constructs an expression of type τ'; this expression is an abstraction of e using an uninterpreted function f_{abs}. We will describe it in more details in Section 3.2.

Each expression is associated with an integer evaluation level (EL) to indicate the order in which this expression should be evaluated. Constants and variables have 0 EL; the EL of a primitive expression equals to the maximum EL of its components. The EL of a quantified expression is higher than those of all its components; analogously for a function application or predicate expression.

$$\zeta(e_1\ \mathtt{op_b}\ e_2) = \mathtt{max}(\zeta(e_1), \zeta(e_2))$$
$$\zeta(\forall v \in [e_1, e_2] : e_3\ v) = \mathtt{max}(\zeta(e_1), \zeta(e_2), \zeta(e_3)) + 1$$
$$\zeta(id(e_1, \ldots, e_n)) = \mathtt{max}(\zeta(e_1), \ldots, \zeta(e_n)) + 1$$

EL values represent the dependency between an expression (*e.g.* a constraint) and its components. When a constraint to be solved, its components are evaluated in the ascending order of their ELs. For

instance, consier the following constraint where a and x are free variables. Suppose the solver cannot handle forall expressions with symbolic ranges, then we will need to convert them into primitive expressions. This forall expression has EL 1 since it uses variable x in its range. To solve this constraint, we first solve the EL 0 constraint $a < x < a+10$ to get a concrete value assignment to a and x, *e.g.* $a = 0$ and $x = 1$; then use these values to instantiate the forall expression, which becomes $\forall v \in [1,1] : v+1 > f(a)$; and then convert this quantified expression to an equivalent one $1+1 > 10+f(a)$, which is obviously satisifiable for $a = 0 \ \wedge \ f(0) = -10$. That is, before solving a constraint with higher EL, we first solve the constraints with lower ELs, then use the obtained concrete values to convert the original costraint to a format that can be handled by the solver. A careful reader may suggest to use an SMT solver capable of handling quantified formulas to handle this example (*e.g.* a simple resolution-based method may work); however KIL supports expressions (*e.g.* recursive functon calls with symbolic inputs) that may be beyond the capacity of typical SMT solvers; and KIL is designed to be a general execution and solving framework not relying on specific advanced solvers. In fact we would like to show in this paper that we can use KIL to define advanced solvers using simple ones.

$$(\forall v \in [1,x] : v+1 > f(a)) \wedge (a < x < a+10)$$

3.2 Function Implementations in KIL

As illustrated in Section 2.1, the execution of the body of a function call can be delayed to the very end. In this case, we only record (in the path condition) some abstract information pertaining to this function, then execute the function body later. This "lazy evaluation" approach can avoid exploring useless paths and states too early.

Now we use the running example in Figure 1 to demonstrate our techniques. In the abstract level we define constraints on the string lengths. Let's first look at the implementation of the find_last_of operation, where *lastSlash* is the index of character c in the string. An abstract constraint (currently we manually identify this constraint; we plan to automate this process using abstraction or summary calculation techniques) is added into the path condition to indicate that *lastSlash* is either npos or an unsigned integer less than the string's length. A predicate expression constraining the string's value is also added.

```
size_type string::find_last_of (char c, size_type pos) {
  size_type lastSlash = klee_create_symbolic_variable(sizeof(size_type), "lastSlash");
  klee_assume(lastSlash == npos || lashSlash < length());
  klee_assume(find_last_of_imp(*this, lastSlash, c, pos));
  return lastSlash;
}
```

There are multiple ways to implement the find_last_of_imp function. The following code uses forall expressions to define the relation between the return value lastSlash and the input arguments including string s. Similar to the printf function in C, klee_assume_IL accepts various argument formats. In this example, for the first klee_assume_IL, KLOVER parses the formatter specified in the first argument, and uses the other arguments to create a forall expression $\forall x \in [0, pos] : s[x] \neq c$ with EL 2. This quantified expression is added into the path constraint.

```
bool find_last_of_imp (size_type lastSlash, string& str, char c, size_type pos = npos) {
  if(pos > len) pos = str.length();
  if (lastSlash == npos)
    klee_assume_IL("@x [0,%0] : %1[x] <> %2", pos, str, c);
  else {
```

```
    klee_assume_IL("@x [%0,%1] : %2[x] <> %3", lastSlash+1, pos, str, c);
    klee_assume(str[lastSlash] == c);
  }
  return true;
}
```

Hence, the execution of the find_last_of function spawns two states with the following path conditions. Note that we may further combine them using an or or ite (if-then-else) expression so that only one path is generated.

state 1	state 2
$lastSlash = npos \wedge$	$lastSlash < len \wedge$
$\forall x \in [0, str.len - 1] : s[x] \neq c$	$\forall x \in [lastSlash + 1, str.len - 1] : str[x] \neq c \wedge str[lastSlash] = c$

Another implementation is to reuse the original uClibc++ implementation to define a relation between the return value lastSlash and the inputs. In the following code, klee_test informs the executor to check whether the given constraint is satisfiable. If yes then the constraint is added into the current path condition; otherwise the current path is terminated without producing any test case. This function is marked _LAZY so that KLOVER will not execute this body implementation when the function is first encountered. Instead this implementation will be used later (*e.g.* when the built-in solver needs it).

```
_LAZY bool find_last_of_imp (size_type lastSlash, string& str, char c, size_type pos = npos) {
  size_type len = str.length();
  if(pos > len) pos = len;
  for(size_type i = pos; i > 0; --i){
    if (str[i-1] == c)
    { klee_test(lastSlash == i - 1); return true; }
  }
  klee_test(lastSlash == npos);
  return true;
}
```

For the call klee_assume(find_last_of_imp(*this, lastSlash, c, pos)) in the find_last_of function shown above, KLOVER creates a predicate expression `find_last_of_imp`$(obj, lastSlash, c, pos)$, where obj refers to the current string object (KLOVER will make this object alive in the executor until this expression is resolved), and $lastSlash$, c and pos refer to their current values respectively. This predicate expression is put into the path condition. Thus, the execution of the find_last_of function leads to only one state with the following path condition.

$$(lastSlash = npos \vee lastSlash < str.len) \wedge$$
$$\texttt{find\_last\_of\_imp}(obj, lastSlash, c, pos)$$

For example, consider the path shown on the left of Figure 3. Its length constraint (after some trivial expression simplifications) is given on the right. Using SMT solving we can figure out the minimum valid value of $str.len$ is 29, which is a valid length. In some cases the derived length is still invalid since the satisfiability of the length constraints alone is not sufficient to guarantee the satisfiability of the original path condition — we will also need to use string value constraints to rule out invalid test case. For example, consider constraints !t.compare(0,4,"www.") and t = "live.com", the length constraints imply that t's minimum length is 8; however it should be 12 when we take into account the characters in string t (*e.g.* $t[0]$ cannot be 'w' and 'l' at the same time). In this case KLOVER will try larger lengths to find out the final valid case.

Path Condition	**Length Constraints**
lastSlash = str.find_last_of('/') $\wedge$	$(lastSlash = npos \vee lastSlash < str.len) \wedge$
lastSlash $\neq$ string::npos $\wedge$	$lastSlash \neq npos \wedge$
rest = str.substr(lastSlash + 1) $\wedge$	$lastSlash + 1 + rest.len \leq str.len \wedge$
rest.find("EasyChair") $\neq$ -1 $\wedge$	$9 \leq rest.len \wedge$
str.compare(0, 7, "http://") $\neq$ 0 $\wedge$	$7 \leq str.len \wedge$
t = str.substr(7, lastSlash-7) $\wedge$	$t.len = lastSlash - 7 \wedge$
!t.compare(0,4,"www") $\wedge$	$4 \leq t.len \wedge$
t' = t.substr(4) $\wedge$	$t'.len + 4 \leq t.len \wedge$
t' = "live.com"	$t'.len = 8$

Figure 3: A path condition (left) and the related length constraints (right) of the running example.

Constraints with Quantified Expressions	**Constraints with Predicate Expressions**
$(\forall x \in [lastSlash+1, str.len) : str[x] \neq 'c') \wedge str[lastSlash] = 'c' \wedge$	find_last_of_imp(lastSlash, str, '/') $\wedge$
$(\forall x \in [0, str.len - lastSlash) : rest[x] = str[lastSlash+1+x]) \wedge$	substr_imp(rest, str, lastSlash + 1) $\wedge$
$(\exists x \in [0, rest.len) : rest[x, x+8] = "EasyChair") \wedge$	find_imp(0, str, "EasyChair") $\wedge$
$(rest[0,7] = "http://") \wedge$	compare_imp(1, str, 0, 7, "http://") $\wedge$
$(t[0, t.len] = str[7, lastSlash]) \wedge$	substr_imp(t, str, 7, lastSlash-7) $\wedge$
$(t[0,3] = "www.") \wedge$	compare_imp(0, str, 0, 4,"www.") $\wedge$
$(t' = t.substr(4)) \wedge$	substr_imp(t', t, 4) $\wedge$
$(t'[0, t.len] = "live.com")$	eq_imp(t', str, "live.com")

Figure 4: Abstract constraints of the path condition shown in Figure 3

Other abstractions schemes. Although here we focus on abstracting strings in terms of their lengths, KIL facilitates defining and using other abstraction schemes in the source code. For example, we can abstract symbolic strings with integers and use these integers to compare the strings. Consider a path condition $s_1 > s_2 \wedge s_2 > s_3 \wedge s_1 < s_3$, where s_1, s_2 and s_3 are symbolic strings. This path condition is obviously unsatisfiable. The abstraction method builds an abstract path condition $\texttt{f}_{\texttt{abs}}(s_1, i32) > \texttt{f}_{\texttt{abs}}(s_2, i32) \wedge \texttt{f}_{\texttt{abs}}(s_2, i32) > \texttt{f}_{\texttt{abs}}(s_3, i32) \wedge \texttt{f}_{\texttt{abs}}(s_1, i32) < \texttt{f}_{\texttt{abs}}(s_3, i32)$, where $\texttt{f}_{\texttt{abs}}(s_i, i32)$ is a KIL expression representing s_i with a 32-bit integer. Specifically, $\texttt{f}_{\texttt{abs}}(s_i, i32)$ represents s_i during string comparisons. With SAT/SMT solving we can quickly obtain the unsat result without needing to execute the body of any string comparison operation. Due to space constraint we will skip describing such abstractions in this paper.

3.3 Abstract Execution with KIL

During the execution, KLOVER may add constraints with various evaluation levels (ELs) into the path conditions. In this section we will describe two methods to solve these constraints. The first one handles quantified formulas while the second one uses lazy function evaluation.

3.3.1 Method I: Solving Abstract Expressions Iteratively

The first one uses iterations to find the solutions. The low EL constraints within a path condition can be evaluated without any delay so as to determine the satisfiability of this path. Since these constraints are a subset of those in the original path condition, when they are unsat, the path condition is also unsat. Conversely, if they are satisfiable, then we can use the witness values to convert those unsolved constraints with higher ELs (*e.g.* quantified formulas), and then solve the expressions generated from

these constraints. We show below the general procedure to process a set of constraints with different ELs.

```
while (true) {
  if (the EL i constraints are unsat or the iteration limit is reached) return unsat;
  else {
    solve these constraints to get witness values for EL i expressions;
    replace these expressions with their values and then convert EL (i+1) constraints;
    if (the resulting constraints are sat) return sat and the witness;
    else {
       negate the conjunction of the assignments to EL i elements;
       add this negation expression into the original set;
       increase the iteration limit by 1;
    }
  }
}
```

Consider the running example. KLOVER first solves the length constraints shown in Figure 3, then uses the length values to instantiate the quantified formulas shown on the left of Figure 4. This converts all quantified formulas to primitive expressions. For example, a valid solution to the length constraints is

$$str.len = 29 \wedge t'.len = 8 \wedge t.len = 12 \wedge rest.len = 9 \wedge lastSlash = 19.$$

This solution is then used to convert the quantified formulas, *e.g.*

$$\forall x \in [lastSlash + 1, str.len) : str[x] \neq \text{`}c'$$

becomes

$$str[19] = \text{`}c' \wedge str[20] \neq \text{`}c' \wedge str[17] \neq \text{`}c' \wedge \ldots \wedge str[28] \neq \text{`}c'.$$

After the conversion of all quantified expressions, the resulting constraints are satisfiable, giving a concrete valid string. If the constraints are unsat, then we will need to start a new iteration after adding the following constraint to rule out the current length assignment and make sure the sum of lengths is increased at least by 1 (this is borrowed from [2] to make the search faster).

Finally a valid solution is found, *e.g.*

$$str.len = 29 \wedge t.len = 12 \wedge rest.len = 9 \wedge lastSlash = 19 \wedge$$
$$str = \text{“}http://www.live.com/EasyChair''\ .$$

Similar iteration methods are used in [2, 5]. One main problem of this method is that it may need many iterations to reach a valid solution or prove the unsatisfiability. What is worse, most iterations may be spent on trying the lengths of intermediate strings such as $t.len$ and $rest.len$. In general, this problem is undecidable; we bound the maximum lengths of strings, and terminate the iteration at time-outs. Hence the method is sound but may be incomplete. Some optimizations may be used to reduce the number of iterations; but we will skip discussing them since our main focus is to show that, by using KIL, we can define a built-in string solver similar to out-of-shelf solvers [2, 5, 8]. In the next section we will present another KIL-based method which needs no explicit iterations, although it also suffers from the incompleteness problem.

3.3.2 Method II: Solving Lazy Function Expressions with Early Termination

Section 3.2 describes a method to use lazy functions to abstract string operations: when a string function is called, we add into the path condition a length constraint together with a predicate characterizing the relation between the input and output of this function. Figure 4 shows a list of predicate expressions for the 7 string operations.

To handle these predicate expressions, we initialize an instance of symbolic execution with two main features: (1) the length constraints are used as an initial assumption before the execution; and (2) the execution terminates immediately whenever a path terminates normally. The first feature enables the executor to use the knowledge on the string lengths to prune trying invalid lengths in the beginning, the second feature makes the executor avoid exploring duplicate paths that lead to no new coverage.

Consider a path with the first four constraints in Figure 3. For the reader's convenience we show below their length constraints and predicate expressions.

Path Condition	**Length Constraints**	**Predicate Expressions**
lastSlash = str.find_last_of('/') ∧	(*lastSlash* = *npos* ∨ *lastSlash* < *str.len*) ∧	find_last_of_imp(lastSlash, str, '/') ∧
lastSlash ≠ string::npos ∧	*lastSlash* ≠ *npos* ∧	
rest = str.substr(lastSlash + 1) ∧	*lastSlash* + 1 + *rest.len* ≤ *str.len* ∧	substr_imp(rest, str, lastSlash + 1) ∧
rest.find("EasyChair") ≠ -1	9 ≤ *rest.len*	find_imp(0, str, "EasyChair")

KLOVER uses a new symbolic execution instance [3] to solve the predicate expressions . First of all, the length constraints are added into the path condition, then a for loop is used to iterate over the length of the input string str. In each iteration the predicate expressions are expanded and executed using their relational implementations. After all the expressions are expanded, if the executor reaches an end state normally (*i.e.* parse all the klee_assume and klee_test checks defined in the function bodies), then it terminates the execution immediately no matter how many unfinished states are left (since we need only one test case for this path).

```
klee_assume(length constraints);
while (str.len < the limit) {
  for (each predicate expression)
    invoke the involved function, execute its body symbolically;
  if (there exists a path that terminate normally)
    return sat and give the witness;
}
return unsat;
```

When the body implementation of a function is executed, the states (paths) failing the klee_test checks will be terminated without producing a test case. Consider the find_last_of_imp function, since the length constraint *lastSlash* ≠ *npos* is enforced, klee_test(lastSlash == npos) will always fail and all paths visiting this statement will be terminated. In contrast, there may be many paths that survive the klee_test(lastSlash == i - 1) check. KLOVER will explore them one by one (*e.g.* use some search heuristics). This will not cause path exploration since KLOVER will stop the execution instance once a path surviving all the checks is found.

```
for(size_type i = str.len; i > 0; --i) {
  if (str[i-1] == 'c') {
    klee_test(lastSlash == i - 1);
    return true; // reach here when str[lastSlash] = 'c' and ∀i > lastSlash : str[i] ≠ 'c'
```

[3] In our implementation, KLOVER associates states with execution modes to distinguish different execution instances.

```
  }
}
klee_test(lastSlash == npos);
return true; // never reach here
```

Similarly the bodies of substr_imp and find_imp are executed to search a valid path passing all the klee_test checks. Some paths are pruned (early), *e.g.* those with path conditions conflicting with the length constraints. Finally a valid input string, *e.g.* "/easychair", is found, and the execution is done although other solutions like ".../easychair" are also possible.

In essence, by using KIL abstraction we can infer information about subsequent executions and use this information to prune useless paths and avoid duplicate paths. A typical way to gather such information is to apply static analysis on the source program [3]. In this paper we show an alternative, *e.g.* do it during symbolic execution. Essentially, KIL supports the implementation of such analyzes or specific solvers directly in the source code, and reuse all facilities built in a high-performance executor such as KLEE. For instance, when processing the length constraints and KIL expressions, KLOVER uses optimizations such as expression rewriting, value concretization, constraint independence, and so on.

4 Evaluation Results

We run KLOVER on some benchmark programs on a laptop with a 2.67GHz Intel Core(TM)i7 processor and 4GB memory. We first test a benchmark program in [2]: the easychair example described in Section 2.1, which is translated by us into C++. Table 1 compares the results for this example when the default uClibc++ implementation, KIL quantified expressions, or KIL predicates are used. Both the value and the length of the input string are symbolic.

For each case, KLOVER achieves much higher performance when using KIL solutions. The early termination trick makes the non-KIL solutions reasonably fast when a path condition is not long, but become much slower when the path is complicated, *e.g.* we observe that more than 100,000 paths are visited when processing Exit 5, path l.

As for the two KIL methods, the quantified one works a little better, especially when a path condition is more complicated. We believe that one main reason is that the underlying SMT solver can prove unsatisfiability faster compared to KLEE's brute-force state searching method. KLEE can tell unsat only after all possible paths are shown to be invalid. This is slower when the path is complicated. Some heuristics (*e.g.* binary search on the lengths, backward-DFS search, *etc.*) may be used to instruct the executor to find the solution faster (not used here though). A good point here is we can use a variety of practical heuristics developed in symbolic executors to speed-up the search.

When the program is run as a whole and all its paths are being visited, the executor will share the states for common prefixes of the paths, making the execution less time-consuming than running each path in a separate execution instance. Besides, if we loosen the requirement of finding the *minimum* lengths, then the performance of KIL solutions can be improved substantially (as indicated in [8]).

We also test KLOVER on various combinations of string operations. Specifically, a set of string operation sequences are generated randomly; each sequence contains at least two string operations. The operations in each sequence are related by data dependency such that the result or side-effect of a previous operation will be used by subsequent operations. Table 2 shows the results for sequences of different lengths (we only try a small set of test cases). Not surprisingly, the longer a sequence is, the better KIL-based methods work. In general, the quantified method works better for larger strings and longer sequences; but the lazy function method is more general and requires less support from the SMT solvers.

Case	Min. Len.	Original Impl.		Start Len.	Method I	Method II
		No Early Term.	+ Early Term.			
Exit (Path) 1	0	<0.1s	<0.1s	0	<0.1s	<0.1s
Exit (Path) 2	1	<0.1s	<0.1s	1	<0.1s	<0.1s
Exit (Path) 3	10	0.1s	<0.1s	10	<0.1s	<0.1s
Imme. Path 1	17	5.2s*	1.8	17	0.1s	0.2s
Imme. Path 2	21	T.O*	T.O	21	0.7s	1s
Exit 4, Path I	17	T.O*	T.O	17	0.2s	0.3s
Exist 5, Path I	29	T.O*	T.O	29	2.1s	2.5s

Table 1: Experimental results on the running example for three approaches in terms of execution time. The first approach uses the original uClibc++ implementation; the second one (Method I) applies iteration-based solving on KIL quantified expressions; and the third one uses lazy function evaluation for KIL predicates. We use "*" to mark the the cases where a fixed length is given to the input string to avoid T.O (*e.g.* $>$ 5 minutes). "Early Term." indicates that the execution instance will terminate once a valid test is found.

Sequence	Method I: Quantified Expr.		Method II: Lazy Func. Eval.	
	small string	large string	small string	large string
seq. len = 4	1-10x	1-100x	1-10x	1-50x
seq. len = 6	1-100x	1-1,000x	1-100x	1-500x
seq. len = 8	1-1,000x	1-1,000x	1-1,000x	1-500x

Table 2: Experimental results on randomly generated string operation sequences. We show the performance improvements (*i.e.* in orders of magnitudes) of KIL-based methods over the one using default uClibc++ implementation (with the early termination optimization). Typically, the lengths of small and large strings are 5-10 and 20-30 respectively.

Discussions. After adopting KIL, KLOVER has also been run on some real C++ programs and achieve encouraging results. These programs use other C++ data structures (*e.g.* C++ containers), and we have defined some initial KIL abstractions for them. We are working on defining better abstractions, and (semi-)automatic ways to find them. The predicate abstraction technique proposed in [3] may help to derive high-quality abstractions automatically. Function summary techniques widely used in symbolic execution may help too. In addition, an interesting approach is to derive a declarative specification from low-level code such as LLVM bytecode, *e.g.* a de-compiler produces specifications from assembly code and gives correctness proof. However, as shown in [10], it often requires manual effort to specify the abstraction or summary for an API function in a non-trivial library; the specification may benefit from a specification language such as KIL. For strings, automaton based approaches [16] will help further reduce the fruitless tries. Note that one key point of our approach is that KIL supports defining these abstractions directly in the source programs, and evaluating them "smartly" in the executor.

Conclusions. We present a declarative intermediate language in KLOVER to assist abstract symbolic execution. It enables KLOVER to avoid exploring many useless or duplicate paths, and define sophisticated built-in solvers in the source code. In particular, we show that it allows flexible and efficient handling of C++ object operations, *e.g.* its lazy function evaluation method and its iteration-based methods as used in modern string solvers. Such a language facilitates the specification of function abstrac-

tion and the control of symbolic execution and constraint solving. Future work includes enhancing the expressiveness of the language, defining abstracts for more C/C++ library classes, developing more optimizations, and testing larger C++ programs. Another key extension is to automate the process of function abstraction (*e.g.* use static analysis or symbolic execution to find function summaries).

References

[1] Saswat Anand, Corina S. Pasareanu & Willem Visser (2007): *JPF-SE: A Symbolic Execution Extension to Java PathFinder*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 134–138.

[2] Nikolaj Bjørner, Nikolai Tillmann & Andrei Voronkov (2009): *Path Feasibility Analysis for String-Manipulating Programs*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 307–321.

[3] Nicolas Blanc, Alex Groce & Daniel Kroening (2007): *Verifying C++ with STL containers via predicate abstraction*. In: *Automated Software Engineering (ASE)*, pp. 521–524.

[4] Cristian Cadar, Daniel Dunbar & Dawson R. Engler (2008): *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 209–224.

[5] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer & Michael D. Ernst (2011): *HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection (Invited Tutorial)*. In: *23rd International Conference on Computer Aided Verification (CAV)*, pp. 1–19.

[6] Chris Lattner & Vikram S. Adve (2004): *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 75–88.

[7] Guodong Li (2011): *Validated Compilation through Logic*. In: *17th International Symposium on Formal Methods (FM)*, pp. 169–183.

[8] Guodong Li, Indradeep Ghosh & Sreeranga P. Rajan (2011): *KLOVER : A Symbolic Execution and Automatic Test Generation Tool for C++ Programs*. In: *23rd International Conference on Computer Aided Verification (CAV)*, pp. 609–615.

[9] Guodong Li, Peng Li, Geof Sawaga, Ganesh Gopalakrishnan, Indradeep Ghosh & Sreeranga P. Rajan (2012): *GKLEE: Concolic Verification and Test Generation for GPUs*. In: *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 215–224. `www.cs.utah.edu/fv/GKLEE`.

[10] Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan & Robert M. Kirby (2010): *Formal Specification of MPI 2.0: Case Study in Specifying a Practical Concurrent Programming API*. *Sci. Comp. Prog.* 75, pp. 65–81.

[11] *The LLVM Compiler Infrastructure* . `http://www.llvm.org/`.

[12] Koushik Sen, Darko Marinov & Gul Agha (2005): *CUTE: a concolic unit testing engine for C*. In: *European Software Engineering Conference/ ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pp. 263–272.

[13] *Satisfiability Modulo Theories Competition (SMT-COMP)*. `http://www.smtcomp.org/2009`.

[14] Nikolai Tillmann & Jonathan de Halleux (2008): *Pex-White Box Test Generation for .NET*. In: *International Conference on Tests and Proofs (TAP)*, pp. 134–153.

[15] *uClibc++: An embedded C++ library*. `http://cxx.uclibc.org`.

[16] Fang Yu, Muath Alkhalaf & Tevfik Bultan (2010): *Stranger: An Automata-Based String Analysis Tool for PHP*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 154–157.

Invariant Generation by Infinite-State Model Checking

Francesco Alberti Natasha Sharygina
Formal Verification and Security Lab.
University of Lugano, Switzerland
{francesco.alberti,natasha.sharygina}@usi.ch

Automatic invariant generation for programs handling unbounded data structures is a long-standing problem in software verification. To generate suitable invariants, the invariant generator needs to be able to infer quantified properties about data structures handled by the program, and at the same time, generate a formula strong enough (the invariant) to prove the correctness of the program, with respect to a given postcondition. In this paper we describe how an infinite-state model checker can be used to generate invariants for programs handling unbounded arrays.

1 Introduction

Checking program correctness means ensuring that a program cannot violate a given property. This is known to be, in its general formulation, an undecidable problem. Nonetheless, many sound (but obviously incomplete) solutions have been proposed so far. In this paper we focus on the *partial correctness* problem for programs, i.e., given a program with a property annotated inside the code at a given location, every execution reaching this location must satisfy the corresponding property.

Program verifiers are tools which take as input programs annotated with properties of interest. They may output a counterexample showing if the program exhibits an execution violating at least one property or a message reporting that the program is correct with respect to them. One well-established way of dealing with program verification [5, 7, 17] is based on the generation of a set of *verification conditions* from the input code. From the analysis of these conditions, the program verifier may be able to check if the code behaves correctly with reference to the annotated properties.

Typically program verifiers are made of three main modules. The first one parses the source code and performs language-dependent analysis tasks. This module outputs a translation of the input code written into an Intermediate Verification Language, which is a simplified language with respect to the source code, with statements for annotating properties previously inferred and properties to be checked. The Intermediate Verification Language is then processed by a tool performing the actual verification task on the code. BOOGIE [5] is one of these tools: it parses a simplified programming language, BoogiePL [14] with *assume* and *assert* statements, encoding known facts about the program and properties to be checked respectively. Verification conditions are generated from this internal representation and given to a theorem prover that checks their (un)satisfiability. The set of program verifiers exploiting BOOGIE capabilities is always growing and more heterogeneous (e.g., DAFNY [32], VCC [11], etc.).

A general way of reasoning in terms of program correctness, underlying also the BOOGIE approach, is by the so-called *Hoare triples* [20, 27],

$$\{P\} \quad S \quad \{Q\} \tag{1}$$

where S is a compound statement representing the input program, P and Q are formulæ over program's variables called *precondition* and *postcondition*, respectively. Checking if (1) is a valid Hoare triple

means checking that if the program S starts in a state satisfying P, it will end in a state satisfying Q. One way to check if (1) is valid is to "propagate" the postcondition Q before S, obtaining another formula Q' [15]. If $P \Rightarrow Q'$, the Hoare-triple (1) is valid. We now restrict our attention to *while loops*: a while loop is a while instruction, i.e. a statement like

$$\mathsf{while}\ C\ \mathsf{do}\ B \tag{2}$$

where B is a (compound) instruction of the language called the *loop body* and C is the *while condition*, and can be thought as a (quantifier-free) formula over the program's variables. In order to check if the Hoare triple (1) is valid when S is a while loop, we need to provide the so called *loop invariant* to the verifier, i.e. a formula H satisfying the following three requirements:

$$\begin{aligned} &\text{(i)} \ \ P \Rightarrow H \\ &\text{(ii)} \ \ \{H \wedge C\}\ \ B\ \ \{H\} \\ &\text{(iii)} \ \ H \wedge \neg C \Rightarrow Q \end{aligned} \tag{3}$$

In other words, this approach requires a formula H which is (i) an overapproximation of the precondition; (ii) if we start executing the loop body from a program state satisfying both H and the while condition, we end up in a state still satisfying H and (iii) if the loop condition is no more satisfied, H is strong enough to prove the post-condition.

Loop invariants are usually suggested by the user: automatic invariant generation for software is a hard task because of the possible infinite number of executions a program might exhibit. To mitigate this problem, abstraction features are usually exploited [12]. *Predicate abstraction* [23] received a lot of attention from the scientific community thanks to its being *context-independent*. States of the programs are clustered according to the predicates they satisfy: this helps in reducing the state space to be explored. Predicate abstraction is usually combined in the CEGAR [9] loop with refinement features, to exclude abstract faulty behaviors with respect to a given property which are infeasible in the concrete model. Different techniques have been used to refine abstract models, e.g., by computation of weakest precondition [4], interpolation [25], etc. In the last decade, an improved version of predicate abstraction has been proposed, called *Lazy abstraction* [26], which allows to achieve different degrees of precision in different parts of the program under verification, and can be efficiently combined with interpolation-based refinement features. The kind of formulæ taken as predicates strongly determines the effectiveness of tools performing the analysis: it is well-known that (alternation of) quantifiers easily lead to undecidability results of satisfiability checks, needed to test if two set of states, i.e., two predicates, have a non-empty intersection or one is entailing the other one. Predicate abstraction at a quantifier-free level is an effective way of performing abstraction of programs handling variables defined over unbounded domains, and has been implemented in many efficient software model checkers (e.g. [6, 8, 10, 36] just to name a few).

Suitable approaches to loop invariants generation for programs handling unbounded data structures have to be able both to infer quantified properties about data of the program and to reason in terms of mathematical induction. Consider for example the program in Figure 1 (taken from [29]). Classical CEGAR-based approaches fail on such program because they infer specific atomic predicates $\mathtt{pc} = 2 \Rightarrow \mathtt{a}[0] = 0$, ..., $\mathtt{pc} = 2 \Rightarrow \mathtt{a}[k] = 0$, while the predicate that is able to stop the divergence is $\mathtt{pc} = 2 \Rightarrow \forall x.(x \geq 0 \wedge x < \mathtt{n} \Rightarrow \mathtt{a}[x] = 0)$. Notice that this piece of code does not have quantified assertions, but the predicate needed to stop divergence does need quantifiers. Different approaches have been studied to infer quantified properties for programs with data structures. On [31] constraints are added to the set of quantified variables as to import and adapt standard predicate abstraction techniques.

```
    var a:[int]int;  var n:int;
    procedure Init ( )
      modifies a;  requires n > 0;
    {
0:    var i:int;  i := 0;
1:    while ( i < n ) {
        a[i] := 0;  i := i+1;
      }
    i := 0;
2:    while ( i < n ) {
        assert( a[i] = 0 );  i:= i+1;
      }
    }
```

Figure 1: Example of a program with assertions without quantified variables.

Template-based predicate inference [39] are effective techniques for inferring predicates, but require user's ingenuity to collect suitable templates. Approaches based on *ghost variables* [19] might not be able to infer predicates where assertions do not contain quantifiers (as the example of Figure 2). Other approaches exploits theorem provers [30], with the limitation of instructing them with axioms to handle arithmetic. Approaches based on range predicates [29] extend interpolating theorem provers [28] by adding inference rules to generate quantified *range predicates* about data stored in arrays handled by input programs. Another approach to generate properties for program with unbounded data structures is abstract interpretation [13, 16, 24].

BOOGIE takes in input programs handling unbounded data structures, such as arrays or more complex data structures. It exploits abstract interpretation-based approaches to infer properties about program data [5]. The outcome of the abstract-interpretation inference engine is then displayed inside of the blocks of the BoogiePL code as assume statements. Logozzo and Leino presented in [34] an approach to strength properties retrieved my means of abstract interpretation to get invariants. Recently [33] another technique has been provided to reason in terms of induction inside a program verifier.

Our contribution In this paper we propose to use an infinite-state model checker to infer invariants for programs handling arrays of unknown length, to address the limitations of the aforementioned approaches. Recently a new model checker called SAFARI (SMT-based Abstraction For Arrays with Interpolants) [3] has been developed. SAFARI integrates a quantified predicate discovering with a backward search algorithm and it is able to check quantified properties for infinite-state transition systems. SAFARI takes as input a transition system and a set of quantified properties and checks if there exists an execution of the system that violates at least one property. If so, a counterexample is returned to the user. On the other hand, if the transition system is "safe" and SAFARI terminates the verification, termination is due to the fact that the model checker built a representation of (an overapproximation of) a set of configurations such that it is not possible to apply the transition relation to an elements inside this set and ending in a state outside it. The contribution of the paper is to show that it is possible to exploit the *proof of safety* returned by SAFARI in order to generate suitable invariants for the input program. What makes SAFARI suitable for this task is the completely declarative approach adapted from the Model Checking Modulo Theories (MCMT) framework [21]. This framework rephrases the problem of model checking infinite-state systems whose states enjoy a well-quasi ordering relation [1, 18] in a completely declarative

```
const L: int;
var a: [int]int;
procedure Find( n:int ) returns ( c:int )
  requires L > 0;
  ensures ((c ≥ L) ⇒ (∀x : int :: (x ≥ 0 ∧ x < L) ⇒ a[x] ≠ n)) ;
{
  c := 0;
  while ( c < L ∧ a[c] ≠ n )
  {
    c := c+1 ;
  }
}
```

Figure 2: BOOGIE program for a Find procedure.

setting, where a particular kind of quantified formulæ can be used as generators of a possibly infinite set of states. The approach behind SAFARI is based on the extension of the lazy abstraction with interpolants framework integrating it with the capabilities of the MCMT one [2].

Outline Section 2 presents *Array-based Transition Systems* (ATS) a suitable formal model for programs handling unbounded arrays with a set of rules to retrieve (ATS) from an imperative program. Section 3 explains how SAFARI checks the safety of an ATS. Section 4 shows how invariants are generated by SAFARI and how BOOGIE can take advantage of them.

2 Array-based Transition Systems

SAFARI is a model checker designed to prove safety (possibly universally quantified) properties of a particular class of infinite-state transition systems called *array-based transition systems* (ATS). We assume the usual first-order syntactic notions of signature, term, formula, and so on. According to [37] a theory is a pair $T = (\Sigma, \mathscr{C})$, where Σ is a set of symbols called the *signature* of T and $\mathscr{C}$ is a class of Σ-structures, the *models* of T. Giving a set of variables $\mathbf{v}$, we represent by $\phi(\mathbf{v})$ a formula where at most variables $\mathbf{v}$ are free. ATS are defined in a completely declarative way over a multi-sorted theory A_I^E, which sort symbols are $\texttt{INDEX}$, $\texttt{ELEM}_\ell$ and $\texttt{ARRAY}_\ell$. Giving a set of variables $\mathbf{v}$, $\mathbf{v}'$ is the set $\mathbf{v}$ where each variables has been renamed by adding a prime; $\mathbf{v}^{(n)}$ is the set $\mathbf{v}$ where every variables has been renamed by adding n primes. An $\exists^I$-formula is a formula $\exists \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}])$ and a $\forall^I$-formula is a formula $\forall \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}])$, where $\underline{i}$ is a tuple of variables of sort $\texttt{INDEX}$ and $\mathbf{a}$ a tuple of variables of sort $\texttt{ARRAY}_\ell$. ATS are defined as pairs $\mathscr{S} = (\mathbf{v}, \tau(\mathbf{v}, \mathbf{v}'))$ where $\mathbf{v}$ is a set of variables and $\tau(\mathbf{v}, \mathbf{v}')$ is a A_I^E-formula encoding the relation between the variables in one state $\mathbf{v}$ and their renamed copies.

A safety problem for SAFARI is a tuple $(\mathscr{S}, I(\mathbf{v}), \{U_k(\mathbf{v})\})$ where $\mathscr{S}$ is an ATS, $I(\mathbf{v})$ is a quantifier-free formula describing the initial state of the transition system and every $U_k(\mathbf{v})$ is a $\exists^I$-formula encoding a bad configuration of $\mathscr{S}$. The safety problem is solved by checking if there exists a A_I^E-satisfiable formula

$$I(\mathbf{v}^{(0)}) \wedge \tau(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau(\mathbf{v}^{(n-1)}, \mathbf{v}^{(n)}) \wedge U_k(\mathbf{v}^{(n)}) \tag{4}$$

for some $n \geq 0$ and some k. SAFARI adapts from the MCMT framework the exploration algorithm, which is a backward reachability procedure starting from unsafe formulæ $U_k(\mathbf{v})$.

```
const L: int;
var a: [int]int;

procedure Find(n: int) returns (c: int);
  requires L > 0;
  ensures c >= L ==> (forall x: int :: x >= 0 && x < L ==> a[x] != n);

implementation Find(n: int) returns (c: int) {
  anon0:
    c := 0;
    goto anon2_LoopHead;
  anon2_LoopHead:  // cut point
    assume {:inferred} 0 <= c;
    goto anon2_LoopDone, anon2_LoopBody;
  anon2_LoopBody:
    assume c < L && a[c] != n;
    c := c + 1;
    goto anon2_LoopHead;
  anon2_LoopDone:
    assume !(c < L && a[c] != n);
    return;
}
```

Figure 3: BoogiePL code for the Find procedure of Figure 2.

2.1 Generating ATS from BOOGIE programs

The first step towards the generation of invariants with SAFARI is the translation from source code to the input language of the model checker. SAFARI takes as input an ATS, which can be generated automatically from the source code. We restrict ourselves to single-procedure programs with integer variables and arrays of integers and annotated with possibly universally quantified assertions and post-conditions. ATS can be generated from these kind of programs starting from their BoogiePL representation. Consider the program in Figure 2. This program consists of a procedure, Find, which takes as input a value n and searches for this value inside an array a of unknown length L. The function returns an index c representing, if $c < L$, the index of a where the first occurrence of n is stored, or, if $c > L$, that n is not stored in any position of a. The BoogiePL program created by BOOGIE for the program presented in Figure 2 is reported in Figure 3. We assume that the instructions in each block of the BoogiePL program created by BOOGIE are independent. This is not ensured in general, but given a BoogiePL program it is easy to retrieve a Control-Flow Graph (CFG) satisfying this assumption. We create an ATS from the CFG computed from a BoogiePL program applying the following five rules. Rules R1 and R2 are used to set-up the tuple of variables $\mathbf{v}$, rules R3, R4 and R5 create the transition relation.

(R1) $\mathbf{v}$ is the set of variables handled by the program.

(R2) L is the set of all the names of the blocks constituting the internal representation of the program plus a new block return. A fresh variable pc ranging over the set of values L is added to $\mathbf{v}$.

(R3) Create one transition for each block of the program:

- assumptions of the block constitute the guard G of the transition representing the block. If a block does not have assumptions, the guard is simply true.
- the literal $\mathtt{pc} = l_s$ is conjoined to the guard of the transition, where l_s is the name of the block.

- if the block has several `goto` blocks, the transition is duplicated and the update $\mathtt{pc}' := l_t$ is added, where l_t is one `goto` block, for all listed `goto` blocks. The `return;` statement is translated into $\mathtt{pc}' := \mathsf{return}$.

(R4) Any guard $G(\cdots \mathtt{a}[\mathtt{i}] \cdots)$ is rewritten to $\exists j.(j = \mathtt{i} \wedge G'(\cdots \mathtt{a}[j] \cdots))$ by introducing a fresh existentially quantified variable j of sort `INDEX`.

(R5) Instructions of the block constitute the update section of the transition. Updates of arrays $\mathtt{a}'[\mathtt{i}] := \mathtt{e}$, where $\mathtt{a}, \mathtt{i}, \mathtt{e} \in \mathbf{v}$ are rewritten as follows:

LHS the guard G is modified as $\exists k.(k = \mathtt{i} \wedge G)$ where k is a fresh variables of sort `INDEX` and the update is changed into $\mathtt{a}' := \lambda j.(\mathsf{if}\ (k = j)\ \mathsf{then}\ \mathtt{e}\ \mathsf{else}\ \mathtt{a}[j])$.[1]

RHS if $\mathtt{e}$ is of the kind $\mathtt{b}[\mathtt{l}]$, where $\mathtt{l}, \mathtt{b} \in \mathbf{v}$, add another fresh existentially quantified variables k' to the transition, add the literal $(k' = \mathtt{l})$ to the guard of the transition and use this new variable k' to index $\mathtt{b}$, i.e., substitute $\mathtt{b}[\mathtt{l}]$ with $\mathtt{b}[k']$.

After these transformations we end up with an ATS $(\mathbf{v}, \tau(\mathbf{v}, \mathbf{v}'))$ where $\tau(\mathbf{v}, \mathbf{v}')$ is a disjunction of guarded assignments in functional form, i.e., formulæ of the kind

$$\exists \underline{k} \begin{pmatrix} G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge \\ \mathbf{a}' = \lambda j.\, U_a(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, a[j]) \wedge \\ \mathbf{s}' = U_s(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix} \tag{5}$$

where $G(\mathbf{v})$ is the guard of the transition that needs to be satisfied in order to execute the transition, and U_a, U_s describe how the variables of the systems are updated by the transition, given the subset $\mathbf{a} \subseteq \mathbf{v}$ of $\mathtt{ARRAY}_\ell$ variables and $\mathbf{s} \subseteq \mathbf{v}$ of `INDEX` and $\mathtt{ELEM}_\ell$ variables. U_a and U_s are tuples of case-defined functions (see [2] for more details). We perform these transformations because it can be shown [21] that the pre-image of a set of configurations represented by an $\exists^I$-formula $C(\mathbf{v})$ with respect to a transition of the kind (5), which is represented by the formula $\exists \mathbf{v}'.(C(\mathbf{v}') \wedge \tau(\mathbf{v}, \mathbf{v}'))$ [38], can be reduce to syntactic manipulation of formulæ, allowing efficient preimage computation. Since we start the computation from $\exists^I$-formulæ, all the formulæ generated by the backward reachability algorithm are all $\exists^I$-formulæ, according to the MCMT framework. Entailment tests, needed to stop the search whenever all states taking the system to the failure have been explored are encoded in $SMT(A_I^E)$ problems. Due to the presence of quantifiers, suitable heuristics are adapted from the MCMT framework [22] in order to handle them in a smart way before asserting the formulæ on the stack of the SMT-Solver. This allows the usage of SMT-Solvers which do not support quantification.

Going ahead with the running example, the Find procedure of Figure 2 will be represented by the following set of transitions:

$\tau_0 := \mathtt{pc} = \mathsf{anon0} \wedge \mathtt{c}' = 0 \wedge \mathtt{pc}' = \mathsf{anon2\_LoopHead}$

$\tau_1 := \mathtt{pc} = \mathsf{anon2\_LoopHead} \wedge \mathtt{c} \geq 0 \wedge \mathtt{pc}' = \mathsf{anon2\_LoopDone}$

$\tau_2 := \mathtt{pc} = \mathsf{anon2\_LoopHead} \wedge \mathtt{c} \geq 0 \wedge \mathtt{pc}' = \mathsf{anon2\_LoopBody}$

$\tau_3 := \exists x.\, \big(\mathtt{pc} = \mathsf{anon2\_LoopBody} \wedge \mathtt{c} < \mathtt{L} \wedge x = \mathtt{c} \wedge \mathtt{a}[x] \neq \mathtt{n} \wedge \mathtt{c}' = \mathtt{c} + 1 \wedge \mathtt{pc}' = \mathsf{anon2\_LoopHead}\big)$

$\tau_4 := \exists x.\, \big(\mathtt{pc} = \mathsf{anon2\_LoopDone} \wedge x = \mathtt{c} \wedge \mathtt{a}[x] = \mathtt{n} \wedge \mathtt{pc}' = \mathsf{return}\big)$

$\tau_5 := \mathtt{pc} = \mathsf{anon2\_LoopDone} \wedge \mathtt{c} \geq \mathtt{L} \wedge \mathtt{pc}' = \mathsf{return}$

The pc variable will take values over the set $L = \{\mathsf{anon0}, \mathsf{anon2\_LoopHead}, \mathsf{anon2\_LoopBody}, \mathsf{anon2\_LoopDone}, \mathsf{return}\}$.

[1] If an update for the array $\mathtt{a}'$ has already been created, simply add a new case to the lambda abstraction.

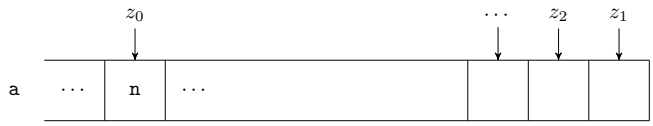

Figure 4: Possible divergence in the Find procedure.

In order to set up a safety problem and let SAFARI run, we need to set up an initial configuration and a set of unsafe formulæ. Initial configuration of the transition system is $\mathtt{pc} = \mathtt{anon0} \wedge \mathtt{L} > 0$ and unsafe configuration is $\neg((\mathtt{pc} = \mathtt{return}) \Rightarrow Q)$, where Q is the post-condition of the procedure, i.e., $Q = \mathtt{c} \geq \mathtt{L} \Rightarrow (\forall x.(x \geq 0 \wedge x < \mathtt{L} \Rightarrow \mathtt{a}[x] \neq n))$.

3 Lazy Abstraction with Interpolants in SAFARI

SAFARI, according to the MCMT framework, explores the state space with iterative algorithms whose goal is to build a collection of configurations $H(\mathbf{v})$. This set represents all the states from which the ATS can reach the violation of one property of the program in a finite number of steps. At the beginning $H(\mathbf{v}) = \bigvee_k U_k(\mathbf{v})$. From $H(\mathbf{v})$ a formula is selected and expanded using $\tau(\mathbf{v}, \mathbf{v}')$. If the new formula $F(\mathbf{v})$ implies $H(\mathbf{v})$ it is deleted, since it represents a set of states already visited; if $F(\mathbf{v}) \wedge I(\mathbf{v})$ is A_I^E-satisfiable it means that there exists an instance of (4) satisfiable, hence there exists a path leading the system from the initial location to the violation of an assertion. Unfortunately, a naïve execution of the described iterative algorithm is going to diverge for imperative programs. Consider again the program Find presented in Figure 2. The backward reachability algorithm will generate the following formulæ:[2]

$$\exists z_0.\,((\mathtt{pc} = \mathtt{return}) \wedge (\mathtt{L} \leq \mathtt{c}) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{L} > z_0))$$
$$\exists z_0, z_1.\,((\mathtt{pc} = \mathtt{anon2\_LoopBody}) \wedge (\mathtt{L} = (\mathtt{c}+1)) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{L} > z_0) \wedge (z_1 = \mathtt{c}) \wedge (\mathtt{n} \neq \mathtt{a}[z_1]))$$
$$\exists z_0, z_1, z_2.\begin{pmatrix}(\mathtt{pc} = \mathtt{anon2\_LoopBody}) \wedge (\mathtt{L} = (\mathtt{c}+2)) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{L} > z_0) \wedge \\ (z_1 = (\mathtt{c}+1)) \wedge (\mathtt{n} \neq \mathtt{a}[z_1]) \wedge (z_2 = \mathtt{c})(\mathtt{n} \neq \mathtt{a}[z_2])\end{pmatrix}$$
$$\ldots$$

and the search will diverge. A graphical representation of what's happening inside SAFARI is reported in Figure 4. SAFARI starts from the unsafe configuration of the program, and then tries to reach the initial location. This is not possible basically because it can always add one position in the array between the current explored cell and the cell indexed by z_0 where it has been assumed the element n is stored.

Abstraction should be used to stop divergence. SAFARI's abstraction features are built according to the Lazy Abstraction paradigm: abstraction of formulæ generated by the backward reachability procedure is performed independently formula by formula: the execution of the backward reachability procedure is interleaved with another procedure that abstracts preimages. If at a certain point the backward procedure generates a consistent formula intersecting with the set of states represented by $I(\mathbf{v})$, then we need to decide the satisfiability of the formula (4). Under suitable hypothesis, this can be done and moreover, if (4) is inconsistent we can retrieve new *quantifier-free* predicates as interpolants to refine the abstract model [2]. Interpolants are then conjoined with the formulæ representing states on the infeasible trace to exclude it from the abstract model.

[2]To simplify the notation we omitted all the literals $z_n \neq z_m$ and $z_n \geq 0$, for every n, m.

3.1 Retrieving quantified predicates and avoiding divergence

This section discusses how SAFARI retrieves quantified predicates and the heuristics we depicted to stop divergence of the interpolation algorithms. It is well-known that interpolation algorithm must be carefully tuned when parameterization comes into play, to prevent their divergence [28].

Notice that the rules (R4) and (R5) executed during the translation from an internal representation of the code into ATS adds quantifiers to the problem. This is the first step towards the inference of quantified predicates. Abstraction-refinement phases can be performed with a heuristic called *term abstraction* [3]. This heuristic takes in input two inconsistent formulæ ϕ_1, ϕ_2 and a list of undesired terms and returns two (still inconsistent) formulæ ϕ_1', ϕ_2' where undesired terms have been substituted with two different fresh constants. This is done to tune the interpolation algorithm: the interpolant for ϕ_1' and ϕ_2' will be likely free from the terms that have been removed from the two formulæ. Consider the formula generated by the backward search executed on the Find program of Figure 2

$$\exists z_0, z_1. ((\mathtt{pc} = \mathsf{anon2\_LoopBody}) \wedge (\mathtt{L} = (\mathtt{c}+1)) \wedge (\mathtt{L} > z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (z_1 = \mathtt{c}) \wedge (\mathtt{n} \neq \mathtt{a}[z_1])) \qquad (6)$$

During the abstraction phase, term abstraction is applied if $\phi(\mathbf{v}) \wedge I(\mathbf{v})$ is inconsistent, where $\phi(\mathbf{v})$ is the newly generated formula. Given $\phi(\mathbf{v})$ and the term abstraction list $\{\mathtt{L}\}$,[3] we apply term abstraction to remove undesired terms from the pair $(\phi(\mathbf{v}), \tilde{I}(\mathbf{v}))$, where $\tilde{I}(\mathbf{v})$ is a formula retrieved from $I(\mathbf{v})$ by adding necessary primed variables.[4] The abstraction computed for formula (6) after term abstraction is

$$\exists z_0, z_1. ((\mathtt{pc} = \mathsf{anon2\_LoopBody}) \wedge ((c+1) > z_0) \wedge (n = a[z_0]) \wedge (z_1 = c) \wedge (\mathtt{n} \neq \mathtt{a}[z_1]))$$

which is more general, and where the two quantified variables are not linked to a precise point of the array. The negation of this formula becomes

$$\mathtt{pc} = \mathsf{anon2\_LoopBody} \Rightarrow \forall z_0, z_1. ((z_0 \leq c) \wedge (z_1 = c) \wedge (\mathtt{a}[z_1] \neq \mathtt{n}) \Rightarrow (\mathtt{a}[z_0] \neq \mathtt{n}))$$

which says that if we are executing the loop body ($\mathtt{pc} = \mathsf{anon2\_LoopBody}$) and $\mathtt{a}[\mathtt{c}]$ does not contain the element we are searching for $\mathtt{n}$, then all previous positions of the array (recall that all z_n's are non-negative and distinct) do not contain $\mathtt{n}$ too.

Term abstraction is also used during the computation of interpolants: terms in the term abstraction list are iteratively removed from pairs of inconsistent formulæ, as to compute an interpolant which will be likely free from these terms too. For the program reported in Figure 1, SAFARI is able to infer the property

$$(\mathtt{pc} = 2) \Rightarrow \forall z_0. ((0 \leq z_0 \wedge \mathtt{n} > z_0) \Rightarrow (\mathtt{a}[z_0] = 0))$$

showing the effectiveness of our approach in inferring useful quantified predicates.

4 Invariant generation with SAFARI

Model checking a system means performing an *exhaustive* exploration of all behaviors of the system. When a model checker reports that the system is safe with respect to a bad configuration U, it returns a *safe inductive invariant* [35], i.e., a set of states H such that (i) the initial set of states of the system is included in H; (ii) if we start from a state in H and we go ahead one step applying the transition relation we end up again in a state inside H, and (iii) the intersection between H and U is empty.

[3] Term abstraction list usually contains iterators or loop bounds. SAFARI has internal features suited for the creation of the term abstraction list.

[4] The reader is referred to [3] for more details.

Rephrased in logical terms, if SAFARI proves that $\mathscr{S}$ is safe with respect to the set $\{U_k(\mathbf{v})\}$, it returns a set of formulæ $H(\mathbf{v})$ such that

$$\begin{aligned} &\text{(i)} \ \bigvee_k U_k(\mathbf{v}) \models_{A_I^E} H(\mathbf{v}) \\ &\text{(ii)} \ \exists \mathbf{v}'(\tau(\mathbf{v},\mathbf{v}') \wedge H(\mathbf{v}')) \models_{A_I^E} H(\mathbf{v}) \\ &\text{(iii)} \ H(\mathbf{v}) \wedge I(\mathbf{v}) \models_{A_I^E} \bot \end{aligned} \tag{7}$$

where $\exists \mathbf{v}'(\tau(\mathbf{v},\mathbf{v}') \wedge H(\mathbf{v}'))$ is the preimage of $H(\mathbf{v})$ with respect to $\tau(\mathbf{v},\mathbf{v}')$ [38]. The safe inductive invariant returned by SAFARI can be added to the BoogiePL code representing the input program to help BOOGIE to conclude the verification.

4.1 Providing Invariants to BOOGIE

Once SAFARI terminates the verification task, and proves that the program is safe, we are left with the collection $H(\mathbf{v})$ which is a disjunction of $\exists^I$-formulæ. We can retrieve an invariant for a BOOGIE program as $\neg H(\mathbf{v})$. For the program in Figure 2 SAFARI selects automatically a suitable term abstraction list and returns an invariant in 0.2 seconds.[5] The invariant we get from SAFARI is the conjunction of the negation of the following formulæ (recall our backward approach and that the invariant we get at the end is a disjunction of $\exists^I$-formulæ): [6]

$$\exists z_0.((\mathtt{pc} = \mathsf{return}) \wedge (\mathtt{L} \leq \mathtt{c}) \wedge (0 \leq z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{L} > z_0))$$

$$\exists z_0.((\mathtt{pc} = \mathsf{anon2\_LoopDone}) \wedge (0 \leq z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{c} > z_0))$$

$$\exists z_0.((\mathtt{pc} = \mathsf{anon2\_LoopHead}) \wedge (0 \leq \mathtt{c}) \wedge (0 \leq z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (\mathtt{c} > z_0))$$

$$\exists z_0, z_1.\begin{pmatrix}(\mathtt{pc} = \mathsf{anon2\_LoopDone}) \wedge (0 \leq z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (z_1 = \mathtt{c}) \wedge \\ (\mathtt{n} = \mathtt{a}[z_1]) \wedge (\mathtt{c} > z_0) \wedge (a[z_0] = a[z_1]) \wedge (z_1 > z_0)\end{pmatrix}$$

$$\exists z_0, z_1.\begin{pmatrix}(\mathtt{pc} = \mathsf{anon2\_LoopBody}) \wedge (\mathtt{L} > \mathtt{c}) \wedge (0 \leq z_0) \wedge (\mathtt{n} = \mathtt{a}[z_0]) \wedge (z_1 = \mathtt{c}) \wedge \\ (\mathtt{n} = \mathtt{a}[z_1]) \wedge ((\mathtt{c}+1) > z_0) \wedge (0 \leq (\mathtt{c}+1)) \wedge (0 < z_1) \wedge ((z_1+1) > z_0)\end{pmatrix}$$

The first one is the (negation of the) postcondition. The other four formulæ can be used to build our invariant by inserting them in the internal representation of the program as further assumptions or directly in the source code. With these four invariants, Boogie is able to verify the program.

5 Experimentation

SAFARI has been tested on various programs handling unbounded arrays. Binaries of the tool and statistics about experiments can be found on SAFARI's website `http://www.verify.inf.usi.ch/safari`. Following the ideas presented in this paper, we plan to fully integrate SAFARI and BOOGIE.

The experiment we will discuss in this section shows the flexibility of our approach with regard to to the invariant generation process. Invariants might be of a different nature, and speak about different facts

[5] Other statistics about this example are: number of nodes generated: 26, SMT-Solver calls: 611, maximum number of `INDEX` variables: 3, CEGAR loops: 6.

[6] This example has been performed with a new version of SAFARI that will be released soon. Output may differ from the one displayed by the available version.

```
var Heap: [int] int;
const unique F: int;   const unique G: int;
const F_final: int;   const G_final: int;
procedure HeapP ( )
  modifies Heap;
  requires F_final > 0 ∧ G_final > 0;
  ensures Heap[F] = F_final ∧ Heap[G] = G_final;
{
  Heap[F] := 0;   Heap[G] := G_final;
  while (Heap[F] < F_final)
    invariant Heap[F] ≤ F_final;
  {
    Heap[F] := Heap[F] + 1;
  }
}
```

Figure 5: Example of a program with a "partial" invariant.

regarding program variables. Consider the following program in Figure 5[7]. Notably, BOOGIE by itself cannot handle the example. In particular, even if the user suggests an invariant, Boogie would complain that the post-condition might not hold. SAFARI, on the contrary, will generate[8] the required invariant. In particular, the invariant generated by SAFARI is made by the negation of the disjunction of the following four formulæ:

$$\exists z_0.\,((\mathtt{pc} = \mathsf{return}) \wedge (z_0 = F) \wedge (\mathtt{F\_final} \neq \mathtt{Heap}[z_0]))$$
$$\exists z_0.\,((\mathtt{pc} = \mathsf{return}) \wedge (z_0 = G) \wedge (\mathtt{G\_final} \neq \mathtt{Heap}[z_0]))$$
$$\exists z_0.\,((\mathtt{pc} = 2) \wedge (z_0 = \mathtt{F}) \wedge (\mathtt{F\_final} \neq \mathtt{Heap}[z_0]) \wedge (\mathtt{F\_final} \leq \mathtt{Heap}[z_0]))$$
$$\exists z_0.\,((\mathtt{pc} = 2) \wedge (z_0 = \mathtt{G}) \wedge (\mathtt{G\_final} \neq \mathtt{Heap}[z_0]))$$

The first two formulæ represent the postcondition. The third formula is the invariant suggested by the user and the forth formula says that in G the array Heap will not be touched by the loop, i.e.,

$$\mathtt{Heap}[\mathtt{G}] = \mathtt{G\_final}$$

With these the additional formula generated by SAFARI the invariant is inductive, and BOOGIE is able to check the input program.

6 Future work

This paper described the overall idea of taking a Boogie program with arrays and exploiting SAFARI to retrieve suitable invariants for it, as to alleviate the burden on users to supply loop invariants. As shown, this method can also cooperate with already inserted invariants, providing support when invariants suggested by the user are not strong enough to prove the correctness of the program.

A fully automatic integration of SAFARI with BOOGIE is an ongoing work. The integration would allow a tight integration of the tools, in such a way that Boogie may pass to SAFARI just a portion of

[7] Thanks to Rustan Leino for this example.

[8] Experimental statistics. Time: 0.06 s, Number of nodes generated: 8, SMT-Solver calls: 199, maximum number of INDEX variables: 3, CEGAR loops: 2.

the program and retrieve invariants in an incremental way. Think about nested loops, the full integration would allow to ask SAFARI for invariants for one loop at a time, starting from the the innermost loop.

Besides the full integration, we believe that many other techniques can be integrated in this technology. First of all, SAFARI can exploit previous inferred properties about the program. These properties can be passed to SAFARI as invariants. The more invariants are given to SAFARI, the better it is: they will help the model checker to converge by pruning the state-space it needs to explorate to converge.

From SAFARI point of view, we observed that invariant might need purification: during the search SAFARI may add quantified variables to create more expressive formulæ. It can be the case that once the invariant has been found, these variables become completely useless when the invariant is generated. The same holds for existentially quantified variables added during the preprocessing steps with rules (R4) and (R5). Invariants generated by SAFARI might have redundant informations. So a post-processing of the invariant before giving it to BOOGIE would be desirable. This post-processing phase would also achieve the goals of returning to BOOGIE a more clear invariant and displaying to the user a more readable formula.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

[2] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, pages 46–61, 2012.

[3] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: SMT-based Abstraction For Arrays with Interpolants. In *CAV*, 2012. To appear.

[4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, pages 158–172, 2002.

[5] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[7] Aaron R. Bradley and Zohar Manna. *Calculus of computation: decision procedures with applications to verification*. Springer, 2007.

[8] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.

[9] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[10] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *TACAS*, pages 570–574, 2005.

[11] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42, 2009.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[13] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.

[14] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[15] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[16] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

[17] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[18] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[19] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[20] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.

[21] Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[22] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[23] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV*, pages 72–83, 1997.

[24] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

[25] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[26] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[28] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[29] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.

[30] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, pages 470–485, 2009.

[31] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[32] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.

[33] K. Rustan M. Leino. Automating induction with an smt solver. In *VMCAI*, pages 315–331, 2012.

[34] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[35] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Temporal verification of reactive systems / Zohar Manna; Amir Pnueli. Springer, 1995.

[36] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[37] Silvio Ranise and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2007.

[38] Tatiana Rybina and Andrei Voronkov. A logical reconstruction of reachability. In *Ershov Memorial Conference*, pages 222–237, 2003.

[39] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.