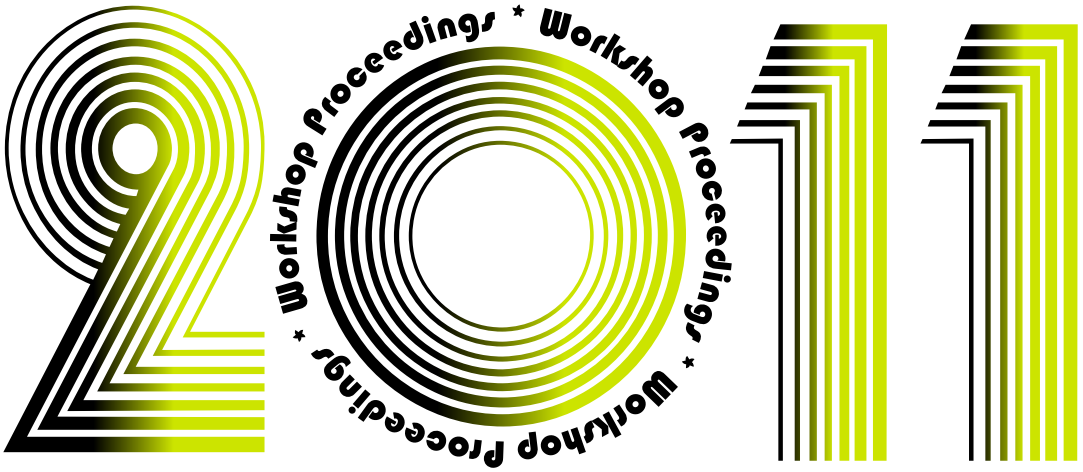
First International Workshop On Intermediate Verification Languages

BOOGIE

BOOGIE

BOOGIE

BOOGIE



@ CADE 23
Wroclaw, Poland

Preface

Software correctness is as important as ever, and with the continuing advances in automatic decision procedures, opportunities for building tools that verify or otherwise analyze programs are greater than ever before. A program verifier is a complex piece of software that, like other software projects, benefits from good software engineering techniques of modularization. By separating concerns, one strives to break the verifier into a number of components, each of which is of manageable complexity. For example, compiler writers have learned to break a compiler up into two major pieces: a front-end that translates a source language into an intermediate representation, and a back-end that translates the intermediate representation into machine code. For a verification tool, an intermediate verification language (IVL) plays an analogous role. The goal of the BOOGIE workshop is to advance the theory and techniques supporting IVLs, to bring together researchers working with IVLs, and to promote sharing of infrastructure that they build.

These informal proceedings contain the papers presented at BOOGIE 2011, the First International Workshop on Intermediate Verification Languages held on 1 August 2011 in Wrocław.

There were 8 submissions. Each submission was reviewed by at least 3 program committee members. The committee decided to accept 7 papers, representing several IVLs, techniques for transforming IVLs, and tools built on IVLs.

20 July 2011
Redmond, WA, USA

K. Rustan M. Leino and
Michał Moskal

Table of Contents

Program Committee

Tayfun Elmas	UC Berkeley
Rustan Leino	Microsoft Research
Claude Marché	INRIA
Michał Moskal	Microsoft Research
Shaz Qadeer	Microsoft Research
Jan Smans	Katholieke Universiteit Leuven
Alexander Summers	ETH Zurich

Automatically Verifying Typing Constraints for a Data Processing Language

Michael Backes[1,2], Cătălin Hriţcu[1,3], Thorsten Tarrach[1,4,5]

[1]Saarland University, Saarbrücken, Germany [2]MPI-SWS, Saarbrücken, Germany
[3]University of Pennsylvania, Philadelphia, USA
[4]Atomia AB, Västerås, Sweden [5]Troxo DOO, Niš, Serbia

Abstract. In this paper we present a new technique for automatically verifying typing constraints in the setting of a first-order data processing language with refinement types and dynamic type-tests. We achieve this by translating programs into a standard while language and then using a general-purpose verification tool. Our translation generates assertions in the while program that faithfully represent the sophisticated typing constraints in the original program. We use a generic verification condition generator together with an SMT solver to prove statically that these assertions succeed in all executions. We formalise our translation algorithm using an interactive theorem prover and provide a machine-checkable proof of its soundness. We provide a prototype implementation using Boogie and Z3 that can already be used to efficiently verify a large number of test programs.

1 Introduction

Dminor [7] is a first-order data processing language with *refinement types* (types qualified by Boolean expressions) and *dynamic type-tests* (Boolean expressions testing whether a value belongs to a type). The combination of refinement types and dynamic type-tests appears in a recent commercial language code-named M [2] and seems to be very useful in practice. However, the increased expressivity allowed by this combination makes statically type-checking programs very challenging.

In this paper we present a new technique for statically checking the typing constraints in Dminor programs by translating these programs into a standard while language. The sophisticated typing constraints in the original program are faithfully encoded as assertions in the generated program and we use a general-purpose verification tool to show statically that these assertions succeed in all executions. This opens up the possibility to take advantage of the huge amount of proven techniques and ongoing research done on general-purpose verification tools.

We have proved that if all assertions succeed in the translated program then the original Dminor program does not cause typing errors when executed. This proof was done in the Coq interactive theorem prover [5], based on a formalisation of our translation algorithm. We thus show formally that, for the language we are considering, a generic verification tool can check the same properties as a sophisticated type-checker. To the best of our knowledge, this is the first machine-checked proof of a translation to an intermediate verification language (IVL).

Finally, we provide a prototype implementation using Boogie and Z3 that can already be used to verify a large number of test programs and we report on an experimental evaluation against the original Dminor type-checker.

1.1 Related work

Biermann et al. [7] were the first to study the combination of refinement types and dynamic type-tests. They introduce a first-order functional language called Dminor, which captures the essence of M [2], but which is simple enough to express formally. They show that the combination of refinement types and dynamic type-tests is highly expressive; for instance intersection, union, negation, singleton, dependent sum, variant and algebraic types are all derivable in Dminor. This expressivity comes, however, at a cost: statically type-checking Dminor programs is very challenging, since the type information can be "hidden" deep inside refinements with arbitrary logical structure. For instance intersection types $T\&U$ are encoded in Dminor as refinement types $(x : \mathsf{Any\ where}\ (x\ \mathsf{in}\ T\ \&\&\ x\ \mathsf{in}\ U))$, where the refinement formula is the boolean conjunction of the results of two dynamic type-tests. Syntax-directed typing rules cannot deal with such "non-structural" types, so Biermann et al. [7] propose a solution based on semantic subtyping. They formulate a semantics in which types are interpreted as first-order logic formulae, subtyping is defined as a valid implication between the semantics of types and they use an SMT solver to discharge such logical formulae efficiently.

The idea of using an SMT solver for type-checking languages with refinement types is quite well established and was used in languages such as SAGE [17], F7 [6], Fine [30] and Dsolve [29]. Biermann et al. [7] show that, in the setting of a first-order language, the SMT solver can play a more central role: They successfully use the SMT solver to check semantic subtyping, not just the refinement constraints. However, while in Dminor [7] subtyping is semantic and checked by the SMT solver, type-checking is still specified by syntax-directed typing rules, and implemented by bidirectional typing rules. In the current work we show that we can achieve very similar results to those of the Dminor type-checker without relying on *any* typing rules, by using the logical semantics of Dminor types directly to generate assertions in a while program, and then verifying the while program using standard verification tools.

Relating type systems and software model-checkers is a topic that has received attention recently from the research community [15, 18, 25]. Our approach is different since we enforce typing constraints using a verification condition generator. Our implementation uses Boogie [20], the generic verification condition generation back-end used by the Verified C Compiler (VCC) [10] and Spec# [4].

There is previous work on integrating verification tools such as Boogie [8] and Why [13] with proof assistants, for the purpose of manually aiding the verification process or proving the correctness of background theories with respect to natural models. However, we are not aware of other machine-checked correctness proofs for translations from surface programming languages into IVLs, even for a language as simple as the one described in this paper. A translation from Java bytecode into Boogie was proved correct in the Mobius project [1, 19], but we are not aware of any mechanized formalisation of this proof.

1.2 Overview

In §2 we provide a brief review of Dminor and in §3 we give a short introduction to our intermediate verification language. §4 and §5 describe our translation algorithm and its implementation. In §6 we compare our work to the Dminor type-checker [7]. Finally, in §7 we conclude and discuss some future work. Further details, our implementation, our Coq formalisation and proofs are all available online at:
`http://www.infsec.cs.uni-saarland.de/projects/dverify`.

2 Review of Dminor (Data Processing Language)

Dminor is a first-order functional language for data processing. We will briefly review this language below; full details are found in the paper by Bierman et al. [7].

Values in Dminor can be simple values (integers, strings, Booleans or null), collections (multi-sets of values) or entities (records). Dminor types include the top type (Any), scalar types (Integer, Text, Logical), collection types ($T*$) and entity types ($\{\ell\colon T\}$). More interestingly, Dminor has *refinement types*: the refinement type $(x : T \text{ where } e)$ consists of the values x of type T satisfying the Boolean expression e.

Syntax of Dminor Expressions:

e ::=	Dminor expression
$x \mid k$	variables and scalar constants
$\oplus(e_1, \ldots, e_n)$	primitive operator application
$e_1?e_2 : e_3$	conditional (if-then-else)
let $x = e_1$ in e_2	let-expression (x bound in e_2)
e in T	dynamic type-test
$\{\ell_i \Rightarrow e_i \ ^{i\in 1..n}\}$	entity (record with n fields $\ell_i \ldots \ell_n$)
$e.\ell$	selects field ℓ of entity e
$\{v_1, \ldots, v_n\}$	collection (multiset)
$e_1 :: e_2$	adding element e_1 to collection e_2
from x in e_1 let $y = e_2$ accumulate e_3	collection iteration (x, y bound in e_3)
$f(e_1, \ldots, e_n)$	function application

Refinement types can be used to express pre- and postconditions of functions, as shown in the type of removeNulls below, where the postcondition states that the resulting collection has at most as many elements as the original collection.

Refinement type used to encode a pre- and postconditions

e.Count $\triangleq$ from x in e let $y = 0$ accumulate $y + 1$

NullableInteger $\triangleq x$: Any where (x in Integer $||$ $x ==$ null)

removeNulls(c : NullableInteger$*$) : (x : Integer$*$ where x.Count $\leq c$.Count) {
  from x in c let $y = \{\}$ accumulate $((x \neq \text{null})?(x :: y) : y)$
}

The dynamic type-test expression e in T pattern-matches the result of expression e against the type T; it returns $\mathbf{true}$ if e has type T and $\mathbf{false}$ otherwise. While dynamic type-test are useful on their own in a data processing language (e.g. for pattern-matching an XML document against a schema represented as a type [14]), they can also be used inside refinement types, which greatly increases the expressivity of Dminor (e.g. it allows encoding union, intersection, negation types, etc., as seen in the example above, where NullableInteger is an encoded union type).

Bierman et al. [7] define a big-step operational semantics for Dminor, in which evaluating an expression can return either a value or "error". An error can for instance arise if a non-existing field is selected from an entity. In Dminor such errors are avoided by the type system, but in this work we rule them out using standard verification tools. The type system by Bierman et al. uses semantic subtyping: they formulate a logical semantics (denotational) in which types are interpreted as first-order logic formulae and subtyping is defined as the valid implication between such formulae. More precisely, they define a function $\mathbf{F}[\![T]\!](v)$ that returns a first-order logic formula testing if the value v belongs to a type T. Since (pure) expressions can appear inside refinement types, $\mathbf{F}[\![T]\!]$ is defined by mutual recursion together with two other functions: $\mathbf{R}[\![e]\!]$ returns a first-order logic term denoting the result[1] of an expression e; and $\mathbf{W}[\![T]\!](v)$ a formula that tests if checking whether v is in type T causes an execution error. The reason for the existence of **W** is that **F** is total and has to return a boolean even when evaluating the expression inside a refinement type causes an error. Our translation makes use of the functions **F** and **W** to faithfully encode the typing constraints in Dminor as assertions in the generated while program.

3 Bemol (Intermediate Verification Language)

We define a simple intermediate verification language (IVL) we call Bemol. Bemol is much simplified compared to a generic IVL: the number of language constructs has been reduced and some Dminor-specific constructs that would normally be encoded were added as primitives. We use Bemol to simplify the presentation, the formalisation of our translation and the soundness proof. In our implementation we use Boogie [3, 11, 20] as the IVL and we encode all Bemol constructs that do not have a direct correspondent in Boogie.

3.1 Syntax and Informal Semantics

Bemol is a while language with collections, records, asserts, mutually recursive procedures, variable scoping and evaluation of logical formulae. The syntax of Bemol is separated into two distinct classes: expressions e, which are side-effect free, and commands c, which have side-effects.

Our expressions allow basic operations on values, most of which directly correspond to the operations in Dminor. Also the available primitive operators $\oplus$ are the

[1] Bierman et al. [7] show that $\mathbf{R}[\![e]\!]$ coincides with the big-step operational semantics on pure expressions – i.e., expressions without side-effects such as non-determinism (accumulate) and non-termination (recursive functions).

same as in Dminor. The only significant difference is the expression formula f which "magically" evaluates the logical formula f and returns a boolean encoding the validity or invalidity of the formula – such a construct is standard in most IVLs. We use the notation $[\![e]\!]_{st}$ for the evaluation of expression e under state st. In case of a typing error (such as selecting a non-existing field from an entity) $\perp$ is returned.

Syntax of Bemol Expressions:

$e ::=$	Bemol expression
x	variable
v	Dminor value (scalar, collection or entity)
$\oplus(e_1, \ldots, e_n)$	primitive Dminor operator application
$e.\ell$	selects field ℓ of entity e
$e_1[\ell := e_2]$	updates field ℓ in entity e_1 with e_2 (produces new entity)
$e_1 :: e_2$	adds element e_1 to collection e_2 (produces new collection)
$e_1 \setminus \{e_2\}$	removes one instance of e_2 from e_1 (produces new collection)
is_empty e	returns **true** if e is the empty collection; **false** otherwise
formula f	returns **true** if formula f is valid in the current state

Syntax of Bemol Commands:

$c ::=$	Bemol command
skip	does nothing
c_1; c_2	executes c_1 and then c_2
$x := e$	assigns the result of e to x
if e then c_1 else c_2	conditional
while e inv a do c end	while loop with invariant a
assert f	expects that formula f holds, causes error otherwise
$x :=$ pick e	puts an element of e in x (non-deterministic)
call P	calls the procedure P
backup x in c	backs up the current state

Bemol commands manipulate the current global state, which is a total function that maps variables to values. The invariant specified in the while command does not affect evaluation; its only goal is to aid the verification condition generator. The pick command chooses non-deterministically an element from collection e and assigns its value to variable x. The call P command transfers control to procedure P, which will also operate on the same global state. The backup x in c command backs up the current state, executes c and once this is finished restores all variables to their former value except for x. This is useful for simulating a call-stack for procedures, and we also introduce it during the translation to simplify the soundness proof. A similar technique is employed by Nipkow [26] for representing local variables. We use this in our encoding for procedure calls below. The encoding uses an entity to pass multiple arguments.

Encoding of procedure calls

x := call P$(e_1, \ldots, e_n)$ $\triangleq$
  backup x in (
    arg := {}; arg := arg$[\ell_1 := e_1]$; ...; arg := arg$[\ell_n := e_n]$; call P; x := ret)
procedure P$(x_1, \ldots, x_n)\{\ c\ \}$ $\triangleq$ proc P $\{\ x_1$:= arg.ℓ_1; ...; x_n := arg.ℓ_n; $c\ \}$

3.2 Operational semantics

We define the big-step semantics of Bemol as a relation $st_{init} \xrightarrow{c} r$, where r can be either a final state st_{final} or **Error**. The only command that can cause an **Error** is the assert command; all the other commands simply "bubble up" the errors produced by failed assertions. If an expression evaluates to $\perp$ it will lead to the divergence of the command that contains it, but this does not cause an error.[2]

3.3 Hoare logics and verification condition generation

We define a Hoare logics for our commands, based on the Software Foundations lecture notes [27] and the ideas of Nipkow [26].

Definition 1 (Hoare triple). *We say that a Hoare triple* $\models \{P\}\, c\, \{Q\}$ *holds semantically if and only if* $\forall st\ r.\ st \xrightarrow{c} r \implies \forall z.\ P\ z\ st = \mathbf{true} \implies \exists st'.\ r = st' \wedge Q\ z\ st' = \mathbf{true}$.

By requiring that the result of the command is not **Error** but an actual state st' we ensure that correct programs do not cause assertions to fail. The meta-variable z is an addition to the traditional Hoare triple and models auxiliary variables, which need to be made explicit in the presence of recursive procedures. Our treatment of auxiliary variables and procedures follows the one of Nipkow [26], who formalises an idea by Morris [24] and Kleymann [16].

The Hoare rules for the standard commands are the same as in Nipkow's work [26], we just list the rules for the constructs that are new to Bemol.

Selected Hoare Rules for Bemol

(Hoare Assert)

$$\frac{}{C \models \{Q \wedge a\}\ \mathsf{assert}\ a\ \{Q\}}$$

(Hoare Pick)

$$\frac{}{C \models \{\lambda z\ st.\ \forall v \in [\![e]\!]_{st}, P\{v/x\}\ z\ st\}\ x\ :=\ \mathsf{pick}\ e\ \{P\}}$$

(Hoare Backup)

$$\frac{\forall st'.\ C \models \{\lambda z\ st.\ P\ z\ st \wedge st' = st\}\ c\ \{\lambda z\ st.\ Q\{st\ x/x\}\ z\ st'\}}{C \models \{P\}\ \mathsf{backup}\ x\ \mathsf{in}\ c\ \{Q\}}$$

[2] Since we only reason about partial-correctness, diverging programs are considered correct. This makes the assumptions on our encoding of Bemol into Boogie be minimal: we only assume that the asserts and successful evaluations of the other commands are properly encoded in Boogie. In §4 we prove that we add enough asserts to capture all errors in the original Dminor program, even under these conservative assumptions we make in the Bemol semantics.

The backup x in c command requires that the Hoare triple for c has the same state for the pre- and postcondition, except for variable x which is updated. We "transfer" the state from the pre- to the postcondition by quantifying over a new state st' that we require to be equal to the state in the precondition.

For our semantics of the Hoare triples it is possible to define a weakest precondition, but not a strongest postcondition function. This is because if c evaluates to **Error** no postcondition is strong enough to make the triple valid. Corresponding to the Hoare rules, we define a verification condition generator (VCgen $c\,Q$), which takes a command c and a postcondition Q as arguments and generates a precondition. We have proved that this is sound, however, the VCgen is not guaranteed to return the weakest precondition, because the user-provided loop invariants are not necessarily the best. The soundness proof of the VCgen crucially relies on the soundness of the Hoare logic rules above.

More importantly for our application, we have proved as a corollary that the programs deemed correct by our VCgen do not cause errors when executed.

Theorem 1 (Soundness of VCgen).
If VCgen $c\,Q$ *returns a valid formula, then* $\nexists st.\ st \xrightarrow{c} \mathbf{Error}$.

4 Translation from Dminor to Bemol

Our translation algorithm is a function $\langle\!\langle\, e \,\rangle\!\rangle_x$ that takes a Dminor program and a variable name x as input and outputs a Bemol program. The variable x is where the generated Bemol program should store the result after it executes. We will introduce the translation using two examples.

In the examples below we consider out to be the variable where the result is put. In Example 1 we show how the removeNulls example from §2 is translated to a while loop that picks and removes elements from the collection until it is empty.

Example 1: Accumulate filtering null **values**

```
removeNulls(c : NullableInteger*) :
  (x : (Integer*) where (x.Count ≤ c.Count))
{
  from x in c let y = {}
    accumulate ((x ≠ null)?(x :: y) : y)
}
```

```
procedure removeNulls(c) {
  assert F⟦NullableInteger*⟧(c);
  y := {};
  c' := c;
  while !is_empty c' inv i(c, c', y) do
    x := pick c';
    c' := c'\{x};
    if x ≠ null then
      y := x :: y
    else
      y := y
  end;
  ret := y;
  assert (F ⟦x : Integer*
    where y.Count ≤ c.Count) ret⟧
}
```

$i(c, c', y) \triangleq \mathbf{F}[\![y : (\mathsf{Integer}*) \text{ where } (c'.\mathsf{Count} + y.\mathsf{Count} \leq c.\mathsf{Count})]\!]\ y$

The loop invariant specifies that the sum of the number of elements in the the intermediate collection c' and the resulting collection y is less or equal than the number of elements in the original collection c. It is not sufficient for the invariant to just reason over y and c as this would be too weak. In this case the invariant is provided by hand on the generated code because this loop invariant is not expressible as a Dminor type. Loop invariant inference on the Dminor side is in general deemed to fail for global properties of collections. Our implementation successfully verifies this example with the provided invariant and in the future we hope to infer such invariants automatically.

As seen in Example 2 for type-tests we first use an assert to check that the type-test does not cause a typing-error and then perform the actual type-test which returns a Logical. Note that **F** is total and would also return a value on a wrongly typed argument.

Example 2: Type-test

x in $(y$: Integer where $y > 5)$	assert $(\neg(\mathbf{W}[\![y$: Integer where $y > 5]\!]\ x))$;
	out := formula $(\mathbf{F}[\![y$: Integer where $y > 5]\!]\ x)$

For illustration, we expand **W** and **F** in the example above; please see the paper by Bierman et al. [7] for the precise definition of these functions of the logical semantics.

$$
\begin{aligned}
&\mathbf{W}[\![y : \text{Integer where } y > 5]\!]\ x \\
\models\ &\mathbf{W}[\![\text{Integer}]\!]\ x \vee \text{let } y = x \text{ in } \neg(\mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{false}) \\
&\quad \vee \mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{true})) \\
\models\ &\mathbf{false} \vee \neg((\text{if } \mathbf{F}[\![\text{Integer}]\!]\ x \text{ then } \mathbf{Return}(x > 5) \text{ else } \mathbf{Error}) = \mathbf{Return}(\mathbf{false}) \\
&\quad \vee (\text{if } \mathbf{F}[\![\text{Integer}]\!]\ x \text{ then } \mathbf{Return}(x > 5) \text{ else } \mathbf{Error}) = \mathbf{Return}(\mathbf{true})) \\
\models\ &\neg\mathbf{F}[\![\text{Integer}]\!]\ x \models \neg(\text{In\_Integer } x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{F}[\![y : \text{Integer where } y > 5]\!]\ x \\
\models\ &\mathbf{F}[\![\text{Integer}]\!]\ x \wedge \text{let } y = x \text{ in } \mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{true}) \\
\models\ &\text{In\_Integer } x \wedge (\text{if In\_Integer } x \text{ then } \mathbf{Return}(x > 5) \text{ else } \mathbf{Error}) = \mathbf{Return}(\mathbf{true}) \\
\models\ &\text{In\_Integer } x \wedge x > 5
\end{aligned}
$$

In case x is not an integer the formula In_Integer $x \wedge x > 5$ is logically equivalent to **false**. Our translation asserts that x is an integer before calling formula in order to match the semantics of Dminor, in which $x > 5$ causes an error when x is not an integer.

4.1 Soundness

We have proved in Coq that if a Dminor program e can raise an error, then the translated program $\langle\!\langle\, e \,\rangle\!\rangle_x$ can evaluate to an error in Bemol. The contrapositive of this is: if the translated program cannot evaluate to an error, then the original Dminor program cannot evaluate to an error either. We have proved this theorem in Coq by induction over the big-step semantics of Dminor $\Downarrow$.

Theorem 2 (Soundness of translation). *If* $e \Downarrow \mathbf{Error}$ *then* $\forall st.\ st \xrightarrow{\langle\!\langle\, e \,\rangle\!\rangle_x} \mathbf{Error}$.

As an immediate consequence of Theorem 1 and Theorem 2 we obtain the soundness of our whole technique.

Corollary 1 (Soundness). *If* VCgen $\langle\!\langle\, e \,\rangle\!\rangle_x$ true *is a valid formula, then* $e \not\Downarrow \mathbf{Error}$.

5 Implementation

Our implementation is called DVerify and translates a Dminor program into a Boogie program. DVerify is written in F# 2.0 [22] and consists of more than 1200 lines of code, as well as a 700 line axiomatisation that defines the Dminor types and functions in Boogie. The Boogie tool then takes the translated Boogie program as input and outputs either an error message that describes points in the program where certain postconditions or assertions may not hold [21] or otherwise prints a message indicating that the program has been verified successfully.

5.1 High-level overview

The heart of our translation algorithm consists of a recursive function that goes over a Dminor expression and translates it into Boogie code. This function is called once per Dminor function and produces a Boogie procedure. Types in Dminor are translated into Boogie function symbols returning bool, using another recursive function in our implementation.

The while loops produced by the translation use the type annotation of accumulate in the source program to generate an invariant for our while loop. In the future we intend to infer such loop invariants automatically using the Boogie infrastructure for this task. The Dminor language as implemented by the type-checker allows for one more construct to define a loop, a from-where-select as in LINQ [23]. In theory from-where-select can be encoded using accumulate, but in the Dminor implementation it is considered primitive in the interest of efficiency and to reduce the type annotation burden. Since from-where-select does not carry a type annotation, we have to find one during translation. For that we use a modified version of the type-synthesis from Dminor that does not call the type-checking algorithm and therefore never fails to synthesise a type for an expression.

We use the Dminor implementation as a library so that we do not have to reimplement existing functionality. This is mainly the parser for Dminor files, the purity checking and a weak form of type-synthesis for from-where-select.

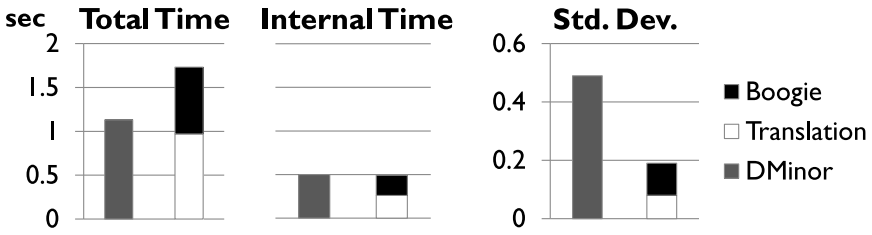
5.2 Axiomatisation

A big part of the implementation is the axiomatisation of Dminor values and functions in Boogie. This is necessary because Boogie as such understands only two sorts, bool and int, whereas Dminor and Bemol have a number of primitive and composite values, such as collections and entities. Our axiomatisation is similar to the axiomatisation the Dminor type-checker feeds to Z3 [7]. In Dminor this axiomatisation is written in SMT-LIB 1.2 syntax [28] and directly fed to Z3 with the proof obligation. Our axiomatisation is in the Boogie language and Boogie translates it to Simplify syntax [12] and feeds it to Z3 along with the verification conditions it generates. Dminor makes heavy usage of the theories Z3 offers, such as extensional arrays and datatypes for example. We use the weak arrays provided by Boogie by default and encode datatypes by hand.

Table 1 Precision comparison

	well-typed	ill-typed
Test suite	76	33
DMinor accepts	66	0
DVerify accepts	62	2 (correct programs)

Chart 1 Speed comparison (average times for 66 well-typed samples)



6 Comparison between Dminor and DVerify

We have tested our implementation against Dminor 0.1.1 from September 2010. Microsoft Research gave us access to their Dminor test suite that contains 109 sample programs. Out of these 109 tests 76 are well-typed Dminor programs and 33 are ill-typed. Out of the 76 well-typed programs the Dminor type-checker cannot verify 10 tests because of incompleteness.

As shown in Table 1, from the 66 cases on which Dminor succeeds, DVerify manages to verify 62 as correct. Out of the 33 that Dminor rejects, DVerify rejects 31. The other two are correct operationally, but are ill-typed with respect to the (inherently incomplete) Dminor type system. Overall this means that DVerify succeeds on 94% of the cases Dminor succeeds on and is able to verify two correct programs Dminor cannot verify. For the 4 correct programs that DVerify cannot verify the most common problem is that type-synthesis generates too complicated loop invariants and Z3 cannot handle the resulting proof obligations.

In order to compare efficiency, we first measured the overall wall-clock time that is needed by the two tools, which includes the time the operating system requires to start the process. Because we are dealing with a large number of small test files and both tools are managed .NET assemblies, initialisation dominates the total running times of both tools. Since initialisation is a constant factor that becomes negligible on bigger examples, we also measured the time excluding initialisation and parsing, which we call "internal time". Chart 1 shows both times (averaged over the 66 well-typed samples accepted by Dminor) on a 2.1 GHz laptop. The internal time is 0.5s on average for both Dminor and DVerify, which means that both tools are very efficient and that our combi-

Table 2 Qualitative comparison of Dminor and DVerify

Area	Our verification approach (DVerify)	Type-checking approach (DMinor)
Verification cond. generation	Weakest precondition	Bidirectional type-checking (type synthesis ≈ strongest postcondition)
Formulae discharged	One per postcondition/assertion (larger, but less obligations)	One per subtyping test (smaller, but more obligations)
Backend	Boogie + SMT-Solver (Z3)	SMT-Solver (Z3)
Loop invariants	In principle Boogie could infer some (even for accumulates) (even for global properties)	For from-where-select (but not for accumulates) (but not for global properties)
Error reporting	Abstract trace	Counterexample
Speed	similar	similar
Precision (practise)	similar	similar
Completeness (theory)	possibly better	possibly worse (type system)
Theories	equality, integers, datatypes, weak arrays	equality, integers, extensional arrays and native encoding of datatypes

nation of a translation and an off-the-shelf verification condition generator matches the average speed of a well-optimised type-checker on its own test suite.

One should keep in mind that all examples in this test suite are relatively small, the biggest one consisting of 90 lines. With bigger examples we expect DVerify to have a speed advantage over Dminor. This is for once due to the much lower standard deviation of DVerify, which indicates a better predictability. It is also reflected in the maximum internal time, where Dminor (3.97s) does three times worse than DVerify (1.35s). The qualitative comparison in Table 2 visualises a summary of this discussion.

7 Conclusion

In this paper we have presented a new technique for statically checking the typing constraints in Dminor programs by translating these programs into a standard while language and then using a general-purpose verification tool.

Future Work

Using a general verification tool for checking the types of Dminor programs should allow us to increase the expressivity of the Dminor language more easily. For example, adding support for *mutable state* would be easy in DVerify: Bemol already supports state, moreover Boogie is used mainly for imperative programming languages [9]. An interesting consequence is that it should be easier to support strong updates in DVerify (i.e. updates that change the type of variables), which is usually quite hard to achieve with a type-checker.

Another very interesting extension is *inferring loop invariants*. Dminor requires that each accumulate expression is annotated with a type for the accumulator which constitutes the invariant of the loop, whereas Boogie has build-in support for abstract interpretation for automatically inferring such invariants [3]. While the invariant inference support in Boogie seems currently very much focused on integer domains, it seems possible to extend it to include support for our Dminor types.

Finally, we would like to improve the *error reporting* capabilities of DVerify. When an assertion fails, Boogie produces an abstract execution trace that outlines which branches were taken to reach the failing assertion and where that assertion is located in the code [21]. In the future we would like to map this trace back to a Dminor trace, and produce errors in terms of the original Dminor program. More interestingly, we would also like to map the partial model produced by the SMT solver when it fails to prove a proof obligation, back as a potential counterexample assignment that maps variables to Bemol values. The Dminor type-checker [7] already implements this and it works quite well, but in Dminor this is implemented from scratch. For DVerify the IVL toolset could in principle provide more support for mapping back the models produced by the SMT solver to something that the end-user can understand.

Acknowledgements We thank Andrew D. Gordon for his helpful comments and the BOOGIE 2011 reviewers for their very useful feedback. Microsoft Research made our work much easier by making the Dminor source code available to us. Cătălin Hriţcu was supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

1. Bytecode level specification language and program logic. Mobius Project, Deliverable D3.1, 2006.
2. The Microsoft code name "M" Modeling Language Specification, October 2009. `http://msdn.microsoft.com/en-us/library/dd548667.aspx`.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects (FMCO)*, Lecture Notes in Computer Science, pages 364–387. Springer, 2005.
4. M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, pages 49–69. Springer, 2005.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual, version 8.2. INRIA, 2009.
6. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 33(2):8, 2011.
7. G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *15th ACM SIGPLAN International Conference on Functional programming (ICFP 2010)*, pages 105–116. ACM Press, 2010.
8. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie - an interactive prover for the Boogie program-verifier. In *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 150–166. Springer, 2008.

9. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.
10. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering (ICSE)*, pages 429–430. IEEE, 2009.
11. R. DeLine and K. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
12. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):473, 2005.
13. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification (CAV)*, pages 173–177. Springer, 2007.
14. H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
15. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. Accepted at CAV, 2011. To appear.
16. T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
17. K. Knowles, A. Tomb, J. Gronski, S. Freund, and C. Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types and `Dynamic`. Technical report, UCSC, 2007.
18. N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *24th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 179–188. IEEE Computer Society, 2009.
19. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science*, 190(1):35–50, 2007.
20. K. R. M. Leino. This is Boogie 2. TechReport, 2008.
21. K. R. M. Leino, T. Millstein, and J. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.
22. C. Marinos. An Introduction to Functional Programming for .NET Developers. *MSDN Magazine*, April 2010.
23. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 706. ACM, 2006.
24. J. Morris. Comments on "procedures and parameters". Undated and unpublished.
25. M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):29, 2008.
26. T. Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
27. B. Pierce, C. Casinghino, M. Greenberg, V. Sjöberg, and B. Yorgey. *Software Foundations*. `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010.
28. S. Ranise and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2006.
29. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.
30. N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. 19th European Symposium on Programming (ESOP 2010)*, pages 529–549, 2010.

Verifying Eiffel Programs with Boogie

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Abstract. Static program verifiers such as Spec#, Dafny, jStar, and VeriFast define the state of the art in automated functional verification techniques. The next open challenges are to make verification tools usable even by programmers not fluent in formal techniques. This paper presents AutoProof, a verification tool that translates Eiffel programs to Boogie and uses the Boogie verifier to prove them. In an effort to be usable with real programs, AutoProof fully supports several advanced object-oriented features including polymorphism, inheritance, and function objects. AutoProof also adopts simple strategies to reduce the amount of annotations needed when verifying programs (e.g., frame conditions). The paper illustrates the main features of AutoProof's translation, including some whose implementation is underway, and demonstrates them with examples and a case study.

1 Usable Verification Tools

It is hard to overstate the importance of *tools* for software verification: tools have practically demonstrated the impact of general theoretical principles, and they have brought automation into significant parts of the verification process. Program provers, in particular, have matured to the point where they can handle complex properties of real programs. For example, provers based on Hoare semantics—e.g., Spec# [2] and ESC/Java [5]—support models of the heap to prove properties of object-oriented applications; other tools using separation logic—e.g., jStar [4] and VeriFast [7]—can reason about complex usages of pointers, such as in the visitor, observer, and factory design patterns. The experience gathered so far has also outlined some design principles, which buttress the development on new, improved verification tools; the success of the Spec# project, for example, has shown the value of using intermediate languages (Boogie [8], in the case of Spec#) to layer a complex verification process into simpler components, which can then be independently improved and reused across different projects.

The progress of verification tools is manifest, but it is still largely driven by challenge problems and examples. While case studies will remain important, verification tools must now also become more practical and *usable* by "lay" programmers. In terms of concrete goals, prover tools should support the complete semantics of their target programming language; they should require minimal annotational effort besides writing ordinary pre and postconditions of routines (methods); and they should give valuable feedback when verification fails.

The present paper describes AutoProof, a static verifier for Eiffel programs that makes some progress towards these goals of increased usability. AutoProof translates Eiffel programs annotated with contracts (pre and postconditions, class invariants, intermediate assertions) into Boogie programs. The translation currently handles sophisticated language features such as exception handling and function objects (called *agents* in Eiffel parlance, and *delegates* in C#). To reduce the need for additional annotations, AutoProof includes simple syntactic rules to generate standard frame conditions from postconditions, so that programmers have to write down explicit frame conditions only in the more complex cases.

This paper outlines the translation of Eiffel programs into Boogie, focusing on the most original features such as exception handling (which is peculiarly different in Eiffel, as opposed to other object-oriented languages such as Java and C#) and the generation of simple frame conditions. The translation of more standard constructs is described elsewhere [18]. At the time of writing, AutoProof does not implement the translation of exceptions described in the paper, but its inclusion is underway. The paper also reports a case study where we automatically verify several Eiffel programs, exercising different language features, with AutoProof. AutoProof is part of EVE (Eiffel Verification Environment), the research branch of the EiffelStudio integrated development environment, which integrates several verification techniques and tools. EVE is distributed as free software and freely available for download at: `http://se.inf.ethz.ch/research/eve/`

Outline. Section 2 presents the Boogie translation of Eiffel's exception handling primitives; Section 3 describes a translation of conforming inheritance that supports polymorphism; Section 4 sketches other features of the translation, such as the definition of "default" frame conditions; Section 5 illustrates the examples verified in the case study; Section 6 presents the essential related work, and Section 7 outlines future work.

2 Exceptions

Eiffel's exception handling mechanism is different than most object-oriented programming languages such as C# and Java. This section presents Eiffel's mechanism (Section 2.1), discusses how to annotate exceptions (Section 2.2), and describes the translation of Eiffel's exceptions to Boogie (Section 2.3) with the help of an example (Section 2.4).

2.1 How Eiffel Exceptions Work

Eiffel exception handlers are specific to each routine, where they occupy an optional **rescue** clause, which follows the routine body (**do**). A routine's **rescue** clause is ignored whenever the routine body executes normally. If, instead, executing the routine body triggers an exception, control is transferred to the **rescue** clause for exception handling. The exception handler will try to restore the object state to a condition where the routine can execute normally. To this end, the body can run more than once, according to the value of an implicit

variable **Retry**, local to each routine: when the execution of the handler terminates, if **Retry** has value **True** the routine body is run again, otherwise **Retry** is **False** and the pending exception propagates to the **rescue** clause of the caller routine.

Figure 1 illustrates the Eiffel exception mechanism with an example. The routine *attempt_transmission* tries to transmit a message by calling *unsafe_transmit*; if the latter routine terminates normally, *attempt_transmission* also terminates normally without executing the **rescue** clause. On the contrary, an exception triggered by *unsafe_transmit* transfers control to the **rescue** clause, which re-executes the body for *max_attempts* times; if all the attempts fail to execute successfully, the attribute (field) *failed* is set and the exception propagates.

```
attempt_transmission (m: STRING)
    local
        failures : INTEGER
    do
        failed := False
        unsafe_transmit (m)
    rescue
        failures := failures + 1
        if failures < max_attempts then
            Retry := True
        else
            failed := True
        end
    end
```

Fig. 1. An Eiffel routine with exception handler.

2.2 Specifying Exceptions

The postcondition of a routine with **rescue** clause specifies the program state both after normal termination and when an exception is triggered. The two post-states are in general different, hence we introduce a global Boolean variable *ExcV*, which is **True** iff the routine has triggered an exception. Using this auxiliary variable, specifying postconditions of routines with exception handlers is straightforward. For example, the postcondition of routine *attempt_transmission* in Figure 1 says that *failed* is **False** iff the routine executes normally:

```
attempt_transmission (m: STRING)
    ensure
        ExcV implies failed
        not ExcV implies not failed
```

The example also shows that the execution of a **rescue** clause behaves as a loop: a routine r with exception handler r **do** s_1 **rescue** s_2 **end** behaves as

the loop that first executes s_1 unconditionally, and then repeats $s_2\,;\, s_1$ until s_1 triggers no exceptions or **Retry** is **False** after the execution of s_2 (in the latter case, s_1 is not executed anymore). To reason about such implicit loops, we introduce a *rescue invariant* [15]; the rescue invariant holds after the first execution of s_1 and after each execution of $s_2\,;\, s_1$. A reasonable rescue invariant of routine *attempt_transmission* is:

rescue invariant
    not *ExcV* **implies not** *failed*
    (*failures* $<$ *max_attempts*) **implies not** *failed*

2.3 Eiffel Exceptions in Boogie

The auxiliary variable *ExcV* becomes a global variable in Boogie, so that every assertion can reference it. The translation also introduces an additional precondition *ExcV* = **false** for every translated routine, because normal calls cannot occur when exceptions are pending, and adds ExcV to the modifies clause of every procedure. Then, a routine with body s_1 and rescue clause s_2 becomes in Boogie:

$\nabla(s_1, excLabel)$
excLabel: **while** (*ExcV*)
        invariant $\nabla(I_{rescue})$;
          {
            ExcV := **false**;
            Retry := **false**;
            $\nabla(s_2, endLabel)$
            if $(\neg$*Retry*$)$ {*ExcV* := **true**; **goto** *endLabel*} ;
            $\nabla(s_1, excLabel)$
          }
endLabel:

where $\nabla(s, l)$ denotes the Boogie translation $\nabla(s)$ of the instruction s, followed by a jump to label l if s triggers an exception:

$$\nabla(s, l) \;=\; \begin{cases} \nabla(s', l)\,;\, \nabla(s'', l) & \text{if } s \text{ is the compound } s'\,;\, s'' \\ \nabla(s)\,;\ \textbf{if}\ (ExcV)\,\{\textbf{goto}\ l;\} & \text{otherwise} \end{cases}$$

Therefore, when the body s_1 triggers an exception, *ExcV* is set and the execution enters the rescue loop. On the other hand, an exception that occurs in the body of s_2 jumps out of the loop and to the end of the routine.

The exception handling semantics is only superficially similar to having control-flow breaking instructions such as *break* and *continue*—available in languages other than Eiffel—inside standard loops: the program locations where the control flow diverts in case of exception are implicit, hence the translation has to supply a conditional jump after every instruction that might trigger an exception. This complicates the semantics of the source code, and correspondingly the verification of Boogie code translating routines with exception handling.

2.4 An Example of Exception Handling in Boogie

Figure 2 shows the translation of the example in Figure 1. To simplify the presentation, Figure 2 renders the attributes *max_attempts*, *failed*, and *transmitted* (set by *unsafe_transmit*) as variables rather than locations in a heap map. The loop in lines 22–36 maps the loop induced by the **rescue** clause, and its invariant (lines 23–24) is the rescue invariant.

3 Inheritance and Polymorphism

The redefinition of a routine r in a descendant class can *strengthen* r's original postcondition by adding an **ensure then** clause, which conjoins the postcondition in the precursor. The example in Figure 3 illustrates a difficulty occurring when reasoning about postcondition strengthening in the presence of polymorphic types. The deferred (abstract) class *EXP* models nonnegative integer expressions and provides a routine *eval* to evaluate the value of an expression object; even if *eval* does not have an implementation in *EXP*, its postcondition specifies that the evaluation always yields a nonnegative value stored in attribute *last_value*, which is set as side effect (see Section 4.1). Classes *CONST* and *PLUS* respectively specialize *EXP* to represent integer (nonnegative) constants and addition. Class *ROOT* is a client of the other classes, and its *main* routine attaches an object of subclass *CONST* to a reference with static type *EXP*, thus exploiting polymorphism.

The verification goal consists in proving that, after the invocation *e.eval* (in class *ROOT*), *eval*'s postcondition in class *CONST* holds, which subsumes the **check** statement in the caller. Reasoning about the invocation only based on the static type *EXP* of the target *e* guarantees the postcondition $last\_value \geq 0$, which is however too weak to establish that *last_value* is exactly 5.

Other approaches, such as Müller's [13], have targeted these issues in the context of Hoare logics, but they usually are unsupported by automatic program verifiers. In particular, with the Boogie translation of polymorphic assignment implemented in Spec#, we can verify the assertion **check** $e.last\_value = 5$ **end** in class *ROOT* only if *eval* is declared *pure*; *eval* is, however, not pure. The Spec# methodology selects the pre and postconditions according to static types for non-pure routines: the call *e.eval* only establishes $e.last\_value \geq 0$, not the stronger $e.last\_value = 5$ that follows from *e*'s dynamic type *CONST*, unless an explicit cast redefines the type *CONST*. The rest of the section describes the solution implemented in AutoProof, which handles contracts of redefined routines.

3.1 Polymorphism in Boogie

The Boogie translation implemented in AutoProof can handle polymorphism appropriately even for non-pure routines; it is based on a methodology for agents [14] and on a methodology for pure routines [3,10]. The rest of the section discusses how to translate postconditions of redefined routines in a way

```
1  var max_attempts: int; var failed:bool; var transmitted:bool;
2
3  procedure unsafe_transmit (m: ref);
4        free requires ExcV = false;
5        modifies ExcV, transmitted;
6        ensures ExcV ⟺ ¬ transmitted;
7
8  procedure attempt_transmission (m: ref);
9        free requires ExcV = false;
10       modifies ExcV, transmitted, max_attempts, failed;
11       ensures ExcV ⟺ failed ;
12
13 implementation attempt_transmission (m: ref)
14 {
15   var failures : int;
16   var Retry: bool;
17   entry:
18        failures := 0; Retry := false;
19        failed := false;
20        call unsafe_transmit (m); if (ExcV) { goto excL; }
21        excL:
22            while (ExcV)
23                invariant ¬ExcV ⟹ ¬ failed ;
24                invariant (failures < max_attempts) ⟹ ¬ failed ;
25            {
26                ExcV := false; Retry := false;
27                failures := failures + 1;
28                if ( failures < max_attempts) {
29                    Retry := true;
30                } else {
31                    failed := true;
32                }
33                if (¬ Retry) {ExcV := true; goto endL;}
34                failed := false
35                call unsafe_transmit (m); if (ExcV) { goto excL; }
36            }
37       endL: return;
38 }
```

Fig. 2. Boogie translation of the Eiffel routine in Figure 1.

```
deferred class EXP
feature
    last_value : INTEGER
    eval
        deferred
        ensure
            last_value ≥ 0
        end
end
```

```
class PLUS inherit EXP feature
    left , right : EXP
    eval do
            left . eval ; right . eval
            last_value := left . last_value +
                          right . last_value
        ensure then
            last_value = left . last_value +
                         right . last_value
        end
invariant
    no_aliasing : left ≠ right ≠ Current
end
```

```
class CONST inherit EXP
feature
    value : INTEGER
    eval
        do
            last_value := value
        ensure then
            last_value = value
        end
invariant
    positive_value : value ≥ 0
end
```

```
class ROOT
feature
    main
        local
            e : EXP
        do
            e := create {CONST}.make (5);
            e . eval
            check e . last_value = 5 end
        end
end
```

Fig. 3. Nonnegative integer expressions.

that accommodates polymorphism, while still supporting modular reasoning. Eiffel also supports *weakening of preconditions* in redefined routines; the translation to Boogie handles it similarly as for postconditions (we do not discuss it for brevity).

The translation of the postcondition of a routine r of class X with result type T (if any) relies on an auxiliary function $post.X.r$:

```
function post.X.r (h1, h2: HeapType; c: ref; res: T) returns (bool);
```

which predicates over two heaps (the pre and post-states in r's postcondition), a reference c to the current object, and the result res. r's postcondition in Boogie references the function $post.X.r$, and it includes the translation $\nabla_{post}(X.r)$ of r's postcondition clause syntactically declared in class X:

```
procedure X.r (Current: ref) returns (Result: T);
  free ensures post.X.r (Heap, old(Heap), Current, Result);
  ensures ∇post(X.r)
```

post.*X*.*r* is a **free ensures**, hence it is ignored when proving *r*'s implementation and is only necessary to reason about usages of *r*.

The function *post*.*X*.*r* holds only for the type *X*; for each class *Y* which is a descendant of *X* (and for *X* itself), an axiom links *r*'s postcondition in *X* to *r*'s strengthened postcondition in *Y*:

axiom (∀ *h1*, *h2*: *HeapType*; *c*: **ref**; *r*: *T* •
  $**type**(*c*) <: *Y* ⟹(*post*.*X*.*r*(*h1*, *h2*, *c*, *r*) ⟹ ∇$_{post}$(*Y*.*r*)));

The function $**type** returns the type of a given reference, hence the postcondition predicate *post*.*X*.*r* implies an actual postcondition ∇$_{post}$(*Y*.*r*) according to *c*'s dynamic type.

In addition, for each redefinition of *r* in a descendant class *Z*, the translation defines a fresh Boogie procedure *Z*.*r* with corresponding postcondition predicate *post*.*Z*.*r* and axioms for all of *Z*'s descendants.

```
1  function post.EXP.eval(h1, h2: HeapType; c: ref) returns (bool);
2
3  procedure EXP.eval(current: ref);
4    free ensures post.EXP.eval(Heap, old(Heap), current, result);
5    ensures Heap[current, last_value] ≥ 0;
6    // precondition and frame condition omitted
7
8  axiom (∀ h1, h2: HeapType; o: ref •
9    $type(o) <: EXP ⟹
10     (post.EXP.eval(h1, h2, o) ⟹ (h1[o, last_value] ≥ 0)) );
11 axiom (∀ h1, h2: HeapType; o: ref •
12   $type(o) <: CONST ⟹
13     (post.EXP.eval(h1, h2, o) ⟹ h1[o, last_value] = h1[o, value]) );
14
15 implementation ROOT.main (Current: ref) {
16     var e: ref;
17   entry:
18       // translation of create {CONST} e.make (5)
19     havoc e;
20     assume Heap[e, $allocated] = false;
21     Heap[e, $ allocated] := true;
22     assume $type(e) =CONST;
23     call CONST.make(e, 5);
24       // translation of e. eval
25     call EXP.eval(e);
26       // translation of check e. last_value = 5 end
27     assert Heap[e, last_value] = 5;
28     return;
29 }
```

Fig. 4. Boogie translation of the Eiffel classes in Figure 3.

3.2 An Example of Polymorphism with Postconditions

Figure 4 shows the essential parts of the Boogie translation of the example in Figure 3. The translation of routine *eval* in lines 3–6 references the function *post.EXP.eval*; the axioms in lines 8–13 link such function to r's postcondition in *EXP* (lines 8–10) and to the additional postcondition introduced in *CONST* for the same routine (lines 11–13). The rest of the figure shows the translation of the client class *ROOT*.

4 Other Features

This section briefly presents other features of the Eiffel-to-Boogie translation.

4.1 Default Frame Conditions

Frame conditions are necessary to reason modularly about heap-manipulating programs, but they are also an additional annotational burden for programmers. In several simple cases, however, the frame conditions are straightforward and can be inferred syntactically from the postconditions. For a routine r, let mod_r denote the set of attributes mentioned in r's postcondition; mod_r is a set of (*reference*, *attribute*) pairs. The translation of Eiffel to Boogie implemented in AutoProof assumes that every attribute in mod_r may be modified (that is, mod_r is r's frame), whereas every other location in the heap is not modified. Since every non-pure routine already includes the whole *Heap* map in its **modifies** clause, the frame condition becomes the postcondition clause:

ensures $(\forall\ o\colon \mathbf{ref},\ f\colon Field\ \bullet\ (o,\ f) \notin mod_r \implies Heap[o,\ f]\ =\ \mathbf{old}(Heap[o,\ f]))$;

To ensure soundness in the presence of inheritance, the translation always uses the postcondition of the original routine definition to infer the frame of the routine's redefinitions.

The frame conditions inferred by AutoProof work well for routines whose postconditions only mention attributes of primitive type. For routines that manipulate more complex data, such as arrays or lists, the default frame conditions are too coarse-grained, hence programmers have to supplement them with more detailed annotations. Extending the support for automatically generated frame conditions is part of future work.

4.2 Routines Used in Contracts Pure by Default

The translation of routines marked as *pure* generates the frame condition **ensures** *Heap* =**old**(*Heap*) which specifies that the heap is unchanged. AutoProof implicitly assumes that every routine used in contracts is pure, and the translation reflects this assumption and checks its validity. While the Eiffel language does not require routines used in contract to be pure, it is a natural assumption which holds in practice most of the times, because the behavior of a program should not rely on whether contracts are evaluated or not. Therefore, including this assumption simplifies the annotational burden and makes using AutoProof easier in practice.

4.3 Agents

The translation of Eiffel to Boogie supports *agents* (Eiffel's name for *function objects* or *delegates*). The translation introduces abstract predicates to specify routines that take function objects as arguments: some axioms link the abstract predicates to concrete specifications whenever an agent is initialized. The details of the translation of agents is described elsewhere [14].

5 Case Study

This section presents the results of a case study applying AutoProof to the verification of the 11 programs listed in Table 1. For each example, the table reports its name, its size in number of classes and lines of code, the length (in lines of code) of the translation to Boogie, the time taken by Boogie to verify successfully the example, and the kind of Eiffel features mostly exercised by the example.

Example 1 is a set of routines presented in Meyer's book [12] when describing Eiffel's exceptions; Example 2 is a set of classes part of the EiffelStudio compiler runtime. To verify them, we extended the original contracts with postconditions to express the behavior when exceptions are triggered, and with rescue invariants (Section 2.2).[1] The most difficult part of verifying these example was inventing rescue invariants. Even when the examples are simple, the rescue invariants may be subtle, because they have to include clauses both for normal and for exceptional termination.

Examples 3–5 target polymorphism in verification. The *Expression* example is described in Section 3. The *Sequence* example models integer sequences with the deferred classes *SEQUENCE*, *MONOTONE_SEQUENCE*, and *STRICT_SEQUENCE*, and their effective descendants *ARITHMETIC_SEQUENCE*, and *FIBONACCI_SEQUENCE*. The *Command* example implements the command design pattern with a deferred class *COMMAND* and effective descendants that augment the postcondition of *COMMAND*'s deferred routine *execute*. The encoding of inheritance described in Section 3 is accurate but it also significantly increases the size of the Boogie translation and correspondingly the time needed to handle it. Since a translation that takes dynamic types into account is not always necessary, future work will introduce an option to have AutoProof translating contracts solely based on the static type of references.

Examples 6–8 use agents and are the same examples as in [14]. The *Formatter* example illustrates the specification of functions taking agents as arguments; the *Archiver* example uses an agent with closed arguments; the *Calculator* example implements the command design pattern using agents rather than subclasses.

Examples 9–11 combine multiple features: a cell class that stores integer values; a counter that can be increased and decreased; a bank account class with

[1] As the implementation in AutoProof of translation of exceptions is currently underway, these two examples were translated by hand.

	Example name	Classes	LOC Eiffel	LOC Boogie	Time [s]	Feature
1.	Textbook OOSC2	1	106	481	2.33	Exceptions
2.	Runtime ISE	4	203	561	2.32	Exceptions
3.	Expression	4	134	752	2.11	Inheritance
4.	Sequence	5	195	976	2.28	Inheritance
5.	Command	4	99	714	2.14	Inheritance
6.	Formatter	3	120	761	2.23	Agents
7.	Archiver	4	121	915	2.07	Agents
8.	Calculator	3	245	1426	9.73	Agents
9.	Cell / Recell	3	154	905	2.09	General
10.	Counter	2	97	683	2.02	General
11.	Account	2	120	669	2.04	General
	Total	**35**	**1594**	**8843**	31.36	

Table 1. Examples automatically verified with AutoProof

clients. These examples demonstrate other features of the translation, such as the usage of default frame conditions.

The source code of the examples is available at `http://se.ethz.ch/people/tschannen/boogie2011_examples.zip`. The experiments ran on a Windows 7 machine with a 2.71 GHz dual core Intel Pentium processor and 2GB of RAM.

6 Related Work

Tools such as ESC/Java [5] and Spec# [2] have made considerable progress towards practical and automated functional verification. Spec# is an extension of C# with syntax to express preconditions, postconditions, class invariants, and non-null types. Spec# is also a verification environment that verifies Spec# programs by translating them to Boogie—also developed within the same project. Spec# works on significant examples, but it does not support every feature of C# (for example, delegates are not handled, and exceptions can only be checked at runtime). Spec# includes annotations to specify frame conditions, which make proofs easier but at the price of an additional annotational burden for developers. To ease the annotational overhead, Spec# adds a default frame condition that includes all attributes of the target object. This solution has the advantage that the frame can change with routine redefinitions to include attributes introduced in the subclasses. AutoProof follows a different approach and tries to rely on standard annotations whenever possible, which impacts on the programs that can be verified automatically.

Spec# has shown the advantages of using an intermediate language for verification. Other tools such as Dafny [9] and Chalice [11], and techniques based on Region Logic [1], follow this approach, and they also rely on Boogie as intermediate language and verification back-end, in the same way as AutoProof does.

Separation logic [16] is an extension of Hoare logic with connectives that define separation between regions of the heap, which provides an elegant approach to reasoning about programs with mutable data structures. Verification environments based on separation logic—such as jStar [4] and VeriFast [7]—can verify advanced features such as usages of the visitor, observer, and factory design patterns. On the other hand, writing separation logic annotations requires considerably more expertise than using standard contracts embedded in the programming language; this makes tools based on separation logic more challenging to use by practitioners.

7 Future Work

AutoProof is a component of EVE, the Eiffel Verification Environment, which combines different verification tools to exploit their synergies and provide a uniform and enhanced usage experience, with the ultimate goal of getting closer to the idea of "verification as a matter of course".

Future work will extend AutoProof and improve its integration with other verification tools in EVE. In particular, the design of a translation supporting the expressive model-based contracts [17] is currently underway. Other aspects for improvements are a better inference mechanism for frame conditions and intermediate assertions (e.g., loop invariants [6]); a support for interactive prover as an alternative to Boogie for the harder proofs; and a combination of AutoProof with the separation logic prover also part of EVE [19].

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *In European Conference on Object Oriented Programming*, ECOOP. Springer-Verlag, 2008.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
3. A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, LNCS. Springer-Verlag, 2007.
4. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *Proceedings of OOPSLA*, pages 213–226, 2008.
5. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
6. C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer, 2010.
7. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS 2010*, 2010.
8. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.
9. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR-16, pages 348–370. Springer-Verlag, 2010.

10. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.
11. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 378–393. Springer-Verlag, 2009.
12. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
13. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
14. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In *Proceedings of TOOLS-EUROPE*, LNCS. Springer, 2010.
15. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE*, 2009.
16. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.
17. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *Proceedings of VSTTE'10*, volume 6217 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2010.
18. J. Tschannen. Automatic verification of Eiffel programs. Master's thesis, Chair of Software Engineering, ETH Zurich, 2009.
19. S. van Staden, C. Calcagno, and B. Meyer. Verifying executable object-oriented specifications with separation logic. In *Proceedings of ECOOP'10*, volume 6183 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2010.

Why Hi-Lite Ada?

Jérôme Guitton, Johannes Kanig and Yannick Moy

AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{guitton,kanig,moy}@adacore.com

Abstract. Use of formal methods in verification activities for critical software development is a promising solution to increase the level of confidence compared to the current practice based on testing, for increasingly complex programs, at a lower cost than the current approach. Concretely, the upcoming standard DO-178C for software development in avionics gives credit to formal verification for supporting verification activities. In project Hi-Lite, we pursue the integration of formal proofs with unit testing, for selected parts of a larger C or Ada software development. This integration relies crucially on a common language of specification between testing and formal proofs, where both share the same assertion semantics. For Ada, this language of specification based on subprogram contracts is part of the upcoming standardized version Ada 2012 of the language. In this paper, we describe the specifics of our translation from Ada to the intermediate verification language Why, noting which features of Why we used in our translation, and from which extensions of Why we could benefit in the future.

1 Introduction

Standards for critical software development define comprehensive verification objectives to guarantee the high levels of dependability we expect of life-critical and mission-critical software. All requirements must be shown to be satisfied by the software, which is a costly activity. In particular, verification of low-level requirements is usually demonstrated by developing unit tests, from which high levels of confidence are only obtained at a high cost. This is the driving force for the adoption of formal verification on an equal footing with testing to satisfy verification objectives. The upcoming DO-178C avionics standard states: *Formal methods [..] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.*

Although there are some areas where formal verification can be applied independently [17], most areas where testing is the main source of evidence today would benefit from an integration of formal verification with existing testing practice. This combination should guarantee a coverage of atomic verifications through formal verification and testing. This is the goal of project Hi-Lite [12], a project aiming at combined unit testing and unit proof of C and Ada programs. The core feature enabling this combination is a language of specification (different in C and Ada) that can be both executed and formally analyzed. In this paper, we are interested only in the Ada part of Hi-Lite.

1.1 Contracts in Ada 2012

The forthcoming version of the Ada standard, called Ada 2012 [2], offers a variety of new features to express properties of programs. The most prominent of these are the *Pre* and *Post* aspects which define respectively the precondition and postcondition of a subprogram[1]. These are defined as Boolean expressions over program variables and functions. Additionally, the expression in a postcondition can refer to the value returned by a function `F` as `F'Result`, and to the value in the pre-state (at the beginning of the call) of any variable `V` as `V'Old`.

The execution model for these aspects is simply to insert assertions at appropriate locations, which raise exceptions when violated. For each reference to the pre-state (`V'Old`), the compiler inserts a shallow copy of the corresponding variable's value at the beginning of the subprogram body.

Expressing properties in contracts is greatly enhanced by the use of if-expressions, case-expressions, quantified-expressions and expression-functions, all defined in Ada 2012. The main objective of these features is verification by testing, based on their executable semantics. In particular, quantified-expressions are always expressed over finite ranges or (obviously finite) containers, with a loop through the range or container as execution model: `for all J in 0 .. 10 => P (J)` is true if-and-only-if the function `P` returns `True` for every argument, starting from 0 up to 10.

1.2 Assertion Based Verification

DO-178 [1] verification activities comprise notably requirement-based verification, which consists in checking that subprograms correctly implement the low-level requirements, and robustness verification, which consists in checking the absence of unintended behavior (like run-time errors) in normal and abnormal modes. As shown by work done at Airbus [17], low-level requirements can be mapped to formal contracts on subprograms, and requirement-based verification can be implemented with unit proofs, where subprogram contracts are formally proved. Various projects using the platforms Frama-C and SPARK have also shown that absence of run-time errors can be proved formally on industrial programs [7, 16].

Both verification of subprogram contracts and absence of run-time errors amount to checking assertions at the level of individual subprograms. In the following, we describe how to prove assertions, irrespective of where these assertions come from. A prerequisite to combining tests and proofs is that both should yield consistent results, *i.e.,* when a test fails, the corresponding assertion should not be provable. This means that assertions should have the same dynamic and static semantics, as emphasized by Chalin [6].

1.3 Using Why to Generate Verification Conditions

Our method is based on the generation of verification conditions (VCs), *i.e.,* first-order formulas whose validity implies the correctness of the program w.r.t. its

[1] Ada terminology uses the term *subprogram* for both functions and procedures.

specification, via a weakest precondition calculus. To generate these VCs, we use the Why verification condition generator (VCG) as intermediate language. Why is the VCG at the heart of the Why platform [10]. It features a small imperative programming language with a syntax similar to ML, and a polymorphic first order logic which serves as annotation language for programs.

Why provides minimal features such as side effects using references (but no aliasing), control structures such as if and while statements, and the possibility to define functions with annotations in the form of pre- and postconditions. It also features an exception mechanism and an effect system, which allows to limit the effects of a function to a certain set of variables. Why enforces alias exclusion by allowing only one level of references, which excludes sharing, and by making use of the effect system to prevent parameter aliasing.

From a user point of view, Why accepts a set of program functions as input, and outputs VCs. These VCs can be generated in the syntax of many different automated provers, including Alt-Ergo [8].

Why programs are written in a syntax close to OCaml. The part of the language used in this paper should be self-explanatory and mainly uses standard constructs such as assignments, sequences, loops, branchings and let-expressions. Logic annotations in programs are introduced with curly braces; curly braces at the beginning (the end) of a function introduce a precondition (a postcondition); curly braces in a loop introduce a loop invariant. Types can also carry annotations, such that the type

```
(x : int) -> { P } unit { Q }
```

denotes the type of functions which take an integer as only argument and return nothing (`unit` is the type whose only value is `void`), with precondition `P` and postcondition `Q`.

Ada is a case insensitive language, but an often-followed convention states that identifiers should start with a capital letter. On the contrary, Why identifiers must be lowercase. In this paper, we benefit from this syntactical difference to distinguish Why and Ada code; additionally, the encoding of an Ada type or variable in Why is its lowercase variant.

2 The Big Picture

2.1 Compilation Chain for Proofs

Project Hi-Lite aims at producing a set of tools and methodologies for applying formal verification to critical software development. GNATprove is the name of the formal verification tool of the Ada toolchain developed in Hi-Lite, which is based on the gnat2why translator from Ada to Why.

A main goal of the tools produced in Hi-Lite is to integrate easily in the usual project structure of Ada developments using the GNAT toolset. This means in particular being able to scale robustly to thousands of files grouped in hundreds of projects, physically hosted in various disks and network locations. Scalability

in this context does not only mean to handle all these situations, but also to produce the expected outcome fast enough, including minimal recompilation after incremental changes to the source code. Robustness means that it should be possible to resume the process of generating and proving VCs correctly no matter what interruptions occur (process killed, network access lost, *etc.*).

The key to such a robust scalable process is to consider each phase leading to the proof of source assertions as a compilation phase, which could be applied in parallel to many different units. The goal of this special compilation is not semantic preservation, but conservative proof-checking: all assertions which should be proved to hold on the source program should correspond to assertion in the target code, and all assertions that are proved in the target code should hold in the source program *whenever a correct execution reaches the point of assertion.* Here is a sketch of the compilation phases:

We start by running the GNAT compiler in a special mode to generate local effects in ALI (Ada Link Information) files from Ada source code. These effects list all variable reads, variable writes and subprogram calls in a subprogram. Then, we run the tool gnat2why, which translates an Ada unit into a set of Why program files, based on the information in ALI files. The Why VC generator then produces a set of VC files from the Why programs, and finally the SMT-prover Alt-Ergo attempts proving automatically these VCs. The tool called GNATprove is responsible for running the whole process through these different phases.

2.2 Proofs in Software Engineering

Another major goal of project Hi-Lite is to integrate formal proofs in the regular verification processes based on dynamic testing. It should be possible to apply proofs successfully to parts of a legacy codebase, without requiring a complete rewrite to follow predefined coding guidelines everywhere. We currently only consider sequential programs, although we could in the future consider supporting restricted forms of tasking, based on the Ravenscar profile for Ada [5].

We have defined a subset of Ada called Alfa, which excludes notably pointers (*access types* in Ada), implicit aliasing and exceptions. In particular, the absence of pointers greatly simplifies formal verification, by removing the need for a memory model, similar to the requirement in SPARK [3]. Notice that implicit aliases could still be present through parameter references. Such code is also automatically identified as not belonging to Alfa.

These restrictions match the constraints of the critical software projects we target, as these typically do not use dynamic memory allocation after initialization, limit the use of pointers and restrict to simple control flow constructs[13]. We provide a slightly modified version of the generic standard containers in Ada

targeted at formal verification [9], which provide an attractive alternative to ad-hoc data structures based on pointers. New developments can use these so-called *formal* containers in Alfa code.

A subprogram specification is in Alfa if its signature and contract are in Alfa. A subprogram body is in Alfa if: (1) its specification is in Alfa; (2) its body only contains constructs that belong to Alfa; (3) its body only mentions variables whose type is in Alfa; (4) it only calls subprograms whose specification is in Alfa. Only subprograms whose body is in Alfa may be proved formally. This partition between subprograms not in Alfa, subprograms whose specification is in Alfa, and functions whose body is in Alfa, allows for a fine-grain application of formal proofs to selected subprograms. The tool GNATprove automatically detects to which category each subprogram belongs, and it only applies the translation to Why to those specifications and bodies belonging to Alfa.

Note that local effects are generated for all functions, including those that are not in Alfa or whose body is not in Alfa. The tool gnat2why computes global effects for a subprogram `S` by aggregating the local effects for `S` and all the subprograms in the downward call-graph closure of `S`, present in the ALI files of the corresponding units. Subprograms not in Alfa may contain reads and writes through pointers to data on the stack or the heap. These are abstracted as reads and writes to a special `Heap` variable. Regarding global variables, only direct reads and writes, as well as reads and writes through references, are considered. All other cases of reads and writes to variables (through a memory address or through a call to a subprogram pointer) are forbidden in the programs we consider.

Note also that, in order to prove subprogram `S` (in Alfa), we require that all bodies of subprograms in the downward call-graph closure of `S` are available (even those of subprograms not in Alfa). This requirement is similar to the one for testing `S` by execution. This is a strong limitation, but it still allows applying formal proofs incrementally on an existing codebase.

2.3 Ada to Why Mapping

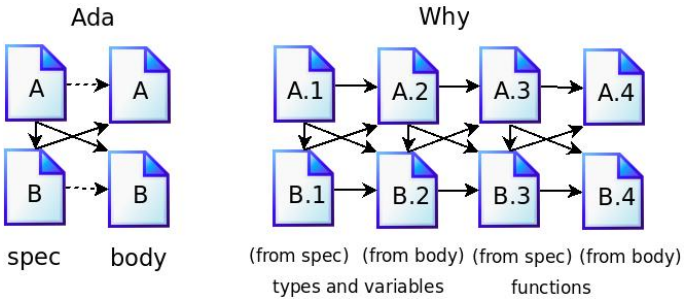
In order to translate Ada code to Why, we had to deal with two major issues: first, Ada units can be mutually dependent on each other, while the graph of dependencies between Why files should form a directed acyclic graph (DAG); secondly, the source units we consider contain both subprograms that can be translated to Why, and subprograms that cannot be translated to Why.

The unit dependency issue means that an Ada unit cannot be simply translated into a Why file. One could consider translating separately the specification and implementation of an Ada unit into two different Why files: subprogram specifications would be translated into Why *function declarations* and subprogram bodies into Why *functions*, with the same Why contract for the function declaration and the function (including global effects) in order to ensure correctness of the approach. This does not work either because the global effects generated for a subprogram declared in the specification may have to mention a global variable declared in the body. Overall, we had to take into account both

intra-unit dependencies and inter-unit dependencies between types, variables and subprograms. Note that simple and mutual recursion between subprograms is just one of these dependencies. In the end, we chose to translate an Ada unit into four Why files:

1. a file for types and variables from the Ada specification;
2. a file for types and variables from the Ada body;
3. a file for subprogram declarations from the unit;
4. a file for subprogram bodies from the unit.

The dependencies between these generated files form a DAG, as depicted below:



Dependencies of a file at level $i \in 2..4$ on a file at a lower level do not introduce circularities; dependencies of a file at level 1 or 3 on a file at the same level follow the non-circular dependencies between Ada unit specifications.

Subprograms whose specification is not in Alfa are not translated into Why. For subprograms whose specification is in Alfa but whose body is not in Alfa, only the subprogram specification is translated into Why.

3 Ada to Why Translation

For the most part, our translation from Ada to Why is similar to existing translations from C or Java to Why [10]. The usual statements (branchings, loops, assignments) and expressions (including the new Ada 2012 expressions) are translated to the corresponding expressions in Why. Arrays are translated as abstract types axiomatized with the theory of functional arrays. Records are translated as abstract types axiomatized with the theory of tuples. Enumerations are translated like integer types. In this section, we take a closer look at the translation of Ada integer types and assertions from contracts and loop invariants.

3.1 Essential Features of Why

There are a number of features of Why that greatly simplify our task of generating correct Why code. The first one is the functional nature of Why, namely the fact that expressions and statements are *not* separated, but can be freely mixed.

This is specially handy when translating those nodes of the GNAT compiler internal tree for Ada expressions which contain statements. It is also convenient to be able to introduce local variables at any time, for example to name intermediate results that should be computed only once.

Why provides a powerful exception mechanism, integrated into the effect system, which allows not only to raise and handle exceptions, but also to specify what a function guarantees in the case of a raised exception. We heavily rely on exceptions, notably to translate loops (see also Section 3.3) and to encode functions with multiple return statements.

The demonic choice operator or any-operator, written `[type]`, returns an arbitrary value of a given type. This is very useful in many situations, for example to translate Ada 2012 quantified expressions (see Section 3.3).

Why allows to introduce labels in programs, denoting program points of interest. In annotations, one can then refer to the value of mutable variables at that point. We plan to use this feature to allow the user to refer to the loop entry point in a loop invariant, for example.

Another crucial feature of Why, of more practical nature, is the ability to track source locations. Each expression and formula of a Why program can be given a name that is associated to a source location of the Ada source program. When generating VCs, Why can now associate each VC to the original Ada source location, and inform about the kind of VC. The possible kinds include preconditions, assertions, and loop invariants. This feature is essential for the generation of helpful error messages.

We do not currently use the type polymorphism provided by Why, with the exception of the `ignore` function introduced in Section 3.3.

3.2 Ada Integer Types, Range and Overflow

The Ada standard defines a few predefined integer types such as `Integer`, `Long_Integer`, *etc.* The exact range of these predefined types depends both on the compiler and the target platform. Usually, they correspond to machine integer types of the underlying architecture, such as 32-bits or 64-bits integers. GNATprove uses the values defined by the GNAT compiler for a given target.

Additionally, the programmer can define his own integer types:

```
type One_Ten is range 1 .. 10;
```

or *derived types* from existing types, using the syntax

```
type One_Ten_Derived is new Integer range 1 .. 10;
```

or *subtypes* of existing types, using the syntax

```
subtype One_Ten_Integer is Integer range 1 .. 10;
```

The two types `One_Ten` and `One_Ten_Derived` are *fresh* types, different from predefined types such as `Integer`; subprograms can be overloaded for these types (*e.g.,* arithmetic operations), and an explicit conversion is needed to transform

a value between these types. The subtype `One_Ten_Integer` does not introduce a new type, so values of type `One_Ten_Integer` are really of type `Integer`.

The three types above have the same range 1 to 10, meaning that a value of such a type should always belong to this range. Any attempt to store a too small or too large value results in a range check failure at run time. This is not to confound with overflow checks, controlled by the *base type*. Each integer type has a corresponding base type, which is the type of intermediate results in an arithmetic expression. Any computation of a too small or too large intermediate value results in an overflow check failure at run time. Both `One_Ten_Derived` and `One_Ten_Integer` share the same base type as `Integer`, which is defined as `Integer` itself. The base type of `One_Ten` depends on the compiler and target, with a minimal range guaranteed of -10 .. 10.

Consider three variables `X`, `Y` and `Z` of type `One_Ten`. Then the assignment

```
Z := (X + Y) / 2;
```

may raise an overflow error, because the sum of `X` and `Y` may exceed the range of the base type, but not a range error, because the overall result of the right hand side will always be in the required range. On the other hand, if `X`, `Y` and `Z` are of type `One_Ten_Derived` or `One_Ten_Integer`, then the assignment

```
Z := X + Y;
```

may not raise an overflow error, because the range of the base type `Integer` is used to carry out the computation. However, if the sum exceeds 10, then the assignment will result in a range error.

Despite the complexity of Ada integer types, we were able encode them into Why like done by Jean-Christophe Filliâtre in the Caduceus tool [10], as follows:

```
type t
predicate t_in_range (x : int) = -128 <= x <= 127

logic t_to_int : t -> int
logic t_of_int : int -> t
parameter t_of_int_ :
   n : int -> { t_in_range (n) } t { t_to_int (result) = n }
parameter t_in_range_ :
   n : int -> { t_in_range (n) } int { result = n }

axiom t_range :
   forall x : t.  t_in_range (t_to_int (x))
axiom t_coerce :
 forall x : int.  t_in_range (x) -> t_to_int (t_of_int (x)) = x
axiom t_unicity :
 forall x, y : t.  t_to_int (x) = t_to_int (y) -> x = y
```

This theory contains conversion functions from and to the type `int` of mathematical integers, a range predicate, and a number of axioms. It is worth noting

that the *logical* conversion functions are considered to be total, while the conversion functions, to be used in the Why program space, have a precondition which limits their use to mathematical integers that are in the range of that type. All values of the Ada integer types are considered to be in the range (axiom `t_range`), and conversion back and forth can be eliminated (axiom `t_coerce`). The function declaration `t_in_range_` is the identity function on mathematical integers, but its argument must be in the range of type `t`; this allows a seamless insertion of overflow checks in arithmetic expressions.

All arithmetic operations are translated to Why using the operations on mathematical integers. Conversions and overflow checks are added as needed and may trigger verification conditions. As an example, assume again the variables `X`, `Y` and `Z` to be of type `One_Ten_Derived`, and the assignment

```
Z := X + Y;
```

Then the corresponding code in Why looks as follows:

```
z := one_ten_derived_of_int_ (integer_in_range_
      (one_ten_derived_to_int (x) + one_ten_derived_to_int (y)));
```

The function calls with the suffix `_` generate VCs corresponding to range and overflow checks in Ada.

3.3 Executable Semantics for Assertions

Assigning the same semantics to assertions in tests and in proofs leads to some differences between our translation and the usual approach, in which assertions are considered as predicates. In particular, we must take into account possible run-time errors during the evaluation of an assertion.

Guarded assertions. Consider a function `Add` with the following specification:

```
function Add (X, Y : One_Ten) return One_Ten
  with Pre => (X + Y < 10);
```

In the following discussion, let us focus on the precondition of the function `Add`. In classical logic, this assertion can be true or false, depending on the assumptions that are available concerning `X` and `Y`. However, if we consider it as an assertion in a program, there are three possible outcomes: the expression can evaluate to one of the Boolean values true or false, but the addition `X+Y` can also overflow, which, in Ada, will trigger an exception.

There are several possibilities to deal with this mismatch. One option is to assimilate failing assertions to false assertions, so that the above assertion not only states that the sum of `X` and `Y` is less than `10`, but also that this sum does not overflow nor underflow. This semantics can be realized by adding these additional assertions to formulas when translating them to logic assertions. Such implicit assertions include both run-time checks and preconditions of functions that are used in specifications. This approach, implementing the *run-time assertion semantics* of formulas, has been chosen for example in ESC/Java2.

In Hi-Lite, we decided not to follow this option, because the presence of implicit assertions that are not explicitly stated in the program text is not suitable in the context of critical software development. Instead, we require the programmer to write guarded assertions, *i.e.,* assertions that cannot raise a run-time error, so that arithmetic operations, array accesses, *etc.*, must be protected in assertions. We check the presence of these guards by generating VCs to prove the absence of run-time errors in assertions. Preconditions must be run-time error free in any context, but local assertions and postconditions can use information that stems from the enclosing context. Dafny [14] implements the same behavior. A more complete discussion is available in a recent survey paper [11].

In practice, we generate such VCs by translating assertions like programs, without needing a special mechanism such as the `is-defined` operator of Chalin [6]. The translation of the function `Add` looks as follows:

```
let add (x : one_ten) (y : one_ten) =
   { true } (* make no assumption at all *)
   ignore
     (one_ten_range_
       (one_ten_to_int (x) + one_ten_to_int (y)) < 10);
   assume { one_ten_to_int (x) + one_ten_to_int (y) < 10 };
   ... (* translated body of function Add *)
```

Note the use of the `assume` statement and the `ignore` function, which can be declared in Why as follows, using polymorphism:

```
logic ignore : 'a -> unit
```

Finally, note that in all generated VCs, `x` and `y` are assumed to be in the range of the type `one_ten`, as stated by the axiom `one_ten_range`.

In this particular example, the overflow check cannot be proved, because the addition can overflow, and GNATprove will report an unproved VC.[2] This example illustrates the major difference between our approach and the one chosen in ESC/Java2 [6]: in our model, the formulas which express that assertions are free from run-time errors always appear as *assertions* in Why, *i.e.,* as formulas to be proved, never as *assumptions*. So, the precondition of the `Add` function is incorrect in GNATprove, while on the contrary it would be *stronger* than the one written by the user in ESC/Java2.

Evaluation order. In logic annotations, the order of evaluation is not important, as logic functions cannot have side effects or provoke an error. However, to reflect the executable semantics, a few precautions have to be taken. The first issue is that, whenever the exact order of evaluation is not defined, it should not influence the result of an expression. Such situations arise for example when evaluating the parameters of a function call. In Hi-Lite, we guarantee that by disallowing global effects in functions (but not in procedures), so that function

[2] This VC will be located at the precondition at the function, and it does not depend on this function being called elsewhere in the program.

calls in the same expression cannot interfere. A more precise analysis would of course be possible. The second issue is to correctly reflect the evaluation order, when defined. As a prominent example, the shortcut operator `and then`, which corresponds to the `&&` operator in C, is translated to an if-then-else statement in programs, and to a simple conjunction in formulas.

Quantified Expressions The semantics of quantified expressions (see Section 1.1) is sufficiently complex to raise a number of questions with regard to formal verification. They act as loops over an integer range or the contents of a container, but they exit early, as soon as the Boolean result is determined. Consider the following pathological example:

```
(for all J in 1 .. 10 => (if J = 5 then J /= 1 / 0 else False))
```

This always returns `False`; The potential run-time error at the fifth iteration is never attained, because the first iteration already yields `False`.

For the proofs, we opt for a stricter, clearer semantics by fixing that the enclosed expression must not raise a run-time error, given *any* loop index in the range. This means that the pathological example above is not provable in GNATprove because we require that the quantified expression does not raise an exception for the entire range.

To that effect, we generate the following code, using the any-operator twice:

```
ignore (let j = [ { } type { range (result) } ] in expr);
[ { } bool { if result then forall (i:type). expr_as_pred } ]
```

Why generates the required checks on the quantified expression and at the same time expresses the contents of the Boolean result, using first-order quantification in Why. Note that the Ada expression is translated twice, once to a Why program expression to obtain the checks, and once to a Why predicate.

Two reviewers of an earlier version of this paper pointed out that it is possible and easy to model the more complex semantics, by adding the additional hypotheses that previous loop iterations did not fail, and returned `True` (in the case of a universal quantification). We still like the simplicity of the current approach, but may change in the future.

Loop assertions. Loop invariants are necessary to prove interesting properties of code containing loops; gnat2why interprets assertions at the beginning of a loop as loop invariants. A naive translation would transform the Ada code

```
while C loop
   pragma Assert (P);
   ...
end loop;
```

into a Why loop whose invariant is simply the predicate `p1`. But such a translation would be incorrect because it does not match the dynamic semantics of assertions. Indeed, loop invariants in Why follow the usual semantics à la Hoare: the invariant must initially hold, regardless of whether the loop executes at all,

and it must hold when exiting the loop. In order to match the dynamic semantics of the Ada program including the assertion, then it should suffice that the assertion holds at the beginning of each loop iteration. This semantics can be achieved in Why using the following translation scheme:

```
if c then
   try
      while true do (* infinite loop *)
         { invariant c and p}
         ...
         if not c then raise Exit
      done
   with Exit -> ();
```

Here, we replace the bounded loop by in infinite loop that exits using an exception. We also protect the loop using an if statement, so that the invariant does not need to hold if the loop is not executed, and we use a `raise` statement to exit the loop, so that the invariant does not need to hold when the loop terminates. Finally, we add the condition `c` to the loop invariant, so that this information is present in VCs generated for the loop body.

In general, loop invariants following this *assertion semantics* are weaker than loop invariants following Hoare semantics, because they need to hold at fewer execution points; as a consequence, they are sometimes slightly easier to write. A potential drawback of assertion semantics is that the VC generator cannot simply forget about the loop body for the rest of the code, but has to check the different possibilities to exit the loop. Why deals nicely with this situation.

4 Conclusion and Future Work

In this paper, we presented a translation from Ada to Why, in a context where only parts of a program are proved, and assertions are assigned the same semantics in formal verification as during execution. Various constructs of the Why language were essential in making this translation easier: functional orientation, demonic choice, exceptions and source location tracking all simplify the work which has to be done to translate Ada to Why.

The next generation of the Why tool, called Why3 [4], provides many features that we believe make it a superior choice as a backend of our GNATprove technology. We are currently preparing a migration to Why3 in the near future.

In particular, the Why3 feature of *theories* will allow much of the Why code that is currently generated programmatically (for example the theory of finite integer types) to be written once and for all in a theory, which can then be *cloned* and specialized. This would simplify the code generator, but also the readability of the generated code. This feature seems to be more powerful than the parametric polymorphism existing in the current version of Why, although both serve the same purpose of writing modular specifications.

Future Work on GNATprove consists in adding features of Ada to the supported Alfa language, and providing means to help the programmer write correct programs and meaningful specifications, such as avoiding trivial assertions, obviously false preconditions, redundant assertions and incomplete contracts.

A challenge is the support of nested arrays and records [15]. Even though modeling records is slightly simpler in a setting without aliasing, where record fields can be modeled by references, this simple approach breaks down when arrays can contain records. One currently envisioned solution is to model arrays of records by records of array fields, and to do so recursively, if necessary.

Acknowledgments. We are grateful to the anonymous reviewers for their helpful comments and suggestions.

References

1. DO-178B: Software considerations in airborne systems and equipment certification, 1982.
2. `http://www.ada-auth.org/standards/12rm/html/RM-TTL.html`.
3. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
4. F. Bobot, J.-C. Filliâtre, A. Paskevich, and C. Marché. Why3: Shepherd your herd of provers. Unpublished, 2011.
5. A. Burns. The Ravenscar profile. Technical report, University of York, 2002. Available at `http://www.cs.york.ac.uk/~burns/ravenscar.ps`.
6. P. Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Trans. Softw. Eng.*, 36:275–287, March 2010.
7. R. Chapman. Industrial experience with SPARK. *SIGAda Ada Letters*, Dec. 2000.
8. S. Conchon and E. Contejean. The Alt-Ergo automatic theorem prover, 2008. `http://alt-ergo.lri.fr/`.
9. C. Dross, J.-C. Filliâtre, and Y. Moy. Correct Code Containing Containers. In *5th International Conference on Tests & Proofs (TAP'11)*, Zurich, June 2011.
10. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590, pages 173–177, Berlin, Germany, July 2007.
11. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01a, University of Central Florida, School of EECS, 2009.
12. `http://www.open-do.org/projects/hi-lite/`.
13. G. J. Holzmann. The power of ten – rules for developing safety critical code. *IEEE Computer*, pages 93–95, June 2006.
14. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
15. D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Trans. Program. Lang. Syst.*, 1:226–244, October 1979.
16. D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C: a case study. *Formal Verification of Object-Oriented Software*, June 2010.
17. J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.

Mutual Summaries: Unifying Program Comparison Techniques

Chris Hawblitzel[1], Ming Kawaguchi[2], Shuvendu K. Lahiri[1], and Henrique Rebêlo[3]

[1] Microsoft Research, Redmond, WA, USA
[2] University of California, San Diego
[3] Federal University of Pernambuco, Brazil

Abstract. In this paper, we formalize *mutual summaries* as a contract mechanism for comparing two programs, and provide a method for checking such contracts modularly. We show that mutual summary checking generalizes equivalence checking, conditional equivalence checking and translation validation. More interestingly, it enables comparing programs where the changes are *interprocedural*. We have prototyped the ideas in SymDiff, a Boogie based language-independent infrastructure for comparing programs.

1 Introduction

The ability to compare two programs statically has applications in various domains. Comparing successive versions of a program for behavioral equivalence across various refactoring and ensuring that bug fixes and feature additions do not introduce compatibility issues, is crucial to ensure smooth upgrades [3, 8, 4]. Comparing different versions of a program obtained after various compiler transformations (*translation validation*) is useful to ensure that the compiler does not change the semantics of the source program [9, 7]. There are two enablers for program comparison compared to the more general problem of (single) program verification. First, one of the two programs serves as an implicit specification for the other program. Second, exploiting simple and automated abstractions for similar parts of the program can lead to greater automation and scalability.

In spite of the recent progress in program comparison, there are several unsatisfactory issues. There appears to be a lack of uniform basis to formalize these techniques — techniques for comparing programs range from the use of verification condition generation [3, 4] to checking *simulation relations* [7]. Most techniques focus on *intraprocedural* transformations of loops [9, 7] and do not extend to interprocedural transformations. The ones that focus on interprocedural transformations use coarse abstractions of procedures as uninterpreted functions [3, 4] and are not extensible.

In this paper, we formalize *mutual summaries* as a uniform basis for comparing two programs in a programming language without loops and jumps (but containing recursive procedures). Mutual summaries generalize single program

contracts such as preconditions and postconditions by relating the summaries of two procedures, usually from two different programs. We provide a sound method for checking mutual summaries modularly. We then describe sound program transformations that can transform unstructured programs (containing loops and jumps) to the language above. We demonstrate that our framework is general enough to capture many existing techniques for comparing programs, including those based on checking simulation relations [7]. Our framework currently lacks the automation provided for specific forms of equivalence checking (e.g. identifying procedures to inline while checking programs with mutually recursive procedures [3], or automatically synthesizing a class of simulation relations for compiler transformations [7]). On the other hand, we show examples of comparing two programs with interprocedural changes for monotonic behavior (§3) and conditional equivalence (§4.2) that are beyond the capabilities of earlier works.

Mutual summaries are currently being incorporated into SymDiff [4], a language-independent framework for comparing programs based on Boogie [1]. Programs in various source languages (C, .NET, x86) are analyzed by translating them to Boogie (e.g. we use Havoc [6] to translate C programs into Boogie). SymDiff provides functionalities to report differences as program traces at the source files along with values of program expressions. As a future direction, we hope to extend the framework to formalize frameworks for comparing classes of concurrent programs [11].

2 Programs

In this section, we present a simple programming language that supports recursive procedures, but does not support loops and unstructured jumps. We will show a way to transform programs with unstructured jumps (including loops) into the language presented in this section (§4.3).

2.1 Syntax and semantics

Figure 1 shows the syntax of the programming and the assertion language. The language supports variables (*Vars*) and various operations on them. The type of any variable $\mathsf{x} \in \mathit{Vars}$ is integer (int). Variables can either be procedure local or global. We denote $G \subseteq \mathit{Vars}$ to be the set of global variables for a program.

Expressions (*Expr*) can be either variables, constants, result of a binary operation in the language (e.g. $+, -$, etc.). Expressions can also be generated by the application of a function symbol U to a list of expressions ($U(e, \ldots, e)$). The expression $\mathsf{old}(e)$ refers to the value of e at the entry to a procedure. *Formula* represents Boolean valued expressions and can be the result of relational operations (e.g. $\{\leq, =, \geq\}$) on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or quantified expressions ($\forall u : \mathsf{int}.\phi$). Formulas can also be the result of applying a relation R to a list of expressions. A $R \in \mathit{Relations}$ represents a relation symbol, some

$$
\begin{array}{lll}
\mathsf{x} \in \mathit{Vars} & & \\
R \in \mathit{Relations} & & \\
U \in \mathit{Functions} & & \\
e \in \mathit{Expr} & ::= & \mathsf{x} \mid c \mid e\ \mathsf{binop}\ e \mid U(e,\ldots,e) \mid \mathsf{old}(e) \\
\phi \in \mathit{Formula} & ::= & \mathsf{true} \mid \mathsf{false} \mid e\ \mathsf{relop}\ e \mid \phi \wedge \phi \mid \neg\phi \\
 & & R(e,\ldots,e) \mid \forall u : \mathsf{int}.\ \phi \\
s \in \mathit{Stmt} & ::= & \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathsf{x} := e \mid \mathsf{havoc}\ \mathsf{x} \\
 & & s; s \mid s \ \diamond\ s \mid \mathsf{x} := \mathsf{call}\ f(e,\ldots,e) \\
p \in \mathit{Proc} & ::= & \mathsf{pre}\ \phi_f\ \mathsf{post}\ \psi_f \\
 & & \mathsf{int}\ f(\mathsf{x}_f : \mathsf{int},\ldots) : \mathsf{ret}_f\ \{\ s\ \}
\end{array}
$$

Fig. 1. A simple programming language.

of which may have specific interpretations. For any expression (or formula) e, $FV(e)$ refers to the variables that appear *free* in e.

A map can be modeled in this language, by introducing two special functions $sel \in \mathit{Functions}$ and $upd \in \mathit{Functions}$; $sel(e_1, e_2)$ selects the value of a map value e_1 at index e_2, and $upd(e_1, e_2, e_3)$ returns a new map value by updating a map value e_1 at location e_2 with value e_3.

The statement skip denotes a no-op. The statement assert ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution of the program *fails*. The statement assume ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution of the program is *blocked*. The assignment statement is standard; havoc x scrambles the value of a variable x to an arbitrary value. $s; t$ denotes the sequential composition of two statements s and t. $s \ \diamond\ t$ denotes a non-deterministic choice to either execute statements in s or t. The $s \ \diamond\ t$ statement along with the assume can be used to model conditional statements; the statement if (e) $\{s\}$ else $\{t\}$ is a syntactic sugar for $\{\mathsf{assume}\ e; s\} \ \diamond\ \{\mathsf{assume}\ \neg e; t\}$.

A procedure $p \in \mathit{Proc}$ has a name f, a set of parameters $\mathit{Params}(f)$ of type int, a return variable ret_f of type int, a body s. Procedure calls are denoted using the call statement. The procedure call can have a side effect by modifying one of the global variables. Procedures can be annotated with *contracts*: a precondition pre ϕ_f and a postcondition post ψ_f. ϕ_f is a formula, such that $FV(\phi_f) \subseteq \mathit{Params}(f) \cup G$; ψ_f is a formula, such that $FV(\psi_f) \subseteq \mathit{Params}(f) \cup G \cup \{\mathsf{ret}_f\}$. We refer to preconditions and postconditions as *single program* contracts in the rest of the paper, to distinguish from contracts relating two programs.

A state σ of a program at a given program location is a valuation of the variables in scope, including procedure parameters, locals and the globals. We omit the definition of an execution as it is quite standard.

For a program annotated with contracts, there are standard methods of transforming them into logical formulas (often referred to as verification conditions) [1]. A procedure call is first replaced by assert ϕ_f; havoc g; assume ψ_f, and the resulting call-free fragment is translated into a formula by variants of weakest-precondition transformer [2]. If the resulting formula is valid (usually

```
int g1;

pre x1 >= 0
void Foo1(int x1){
     if (x1 < 100){
        g1 = g1 + x1;
        Foo1(x1 + 1);
     }
}
```

```
int g2;

pre x2 >= 0
void Foo2(int x2){
     if (x2 < 100){
        g2 = g2 + 2*x2;
        Foo2(x2 + 1);
     }
}
```

Fig. 2. Example demonstrating mutual summaries.

checked by a Satisfiability Modulo Theories (SMT) solver [10]), then the program satisfies its contracts.

3 Mutual summaries

3.1 Definition

A *program* P consists of a set of procedures $\{f_1, \ldots, f_k\}$. For the sake of this paper, we assume that programs are *closed*; i.e., if a procedure $f_1 \in P$ calls a procedure f_2, then $f_2 \in P$.

We use the notation $C = \lambda f^1.f^2.\ \phi(f^1, f^2)$ to be an indexed (by a pair of procedures) set of formulas such that $C(f^1, f^2)$ denotes the formula for the pair (f^1, f^2). We extend this notation to refer to an indexed set of expressions, constants, sets of states, *etc.* Hereafter, for simplicity of exposition, we assume that each procedure f takes a single parameter x_f. Unless otherwise mentioned, g refers to the only global variable in a program that can be read and written to by any procedure.

Definition 1 (Mutual summaries). *For any pair of procedures $f^1 \in P^1$ and $f^2 \in P^2$, a formula $C(f^1, f^2)$ is a mutual summary, if the signature of $C(f^1, f^2)$ only refers to variables that are in scope at exit from either f^1 or f^2. This includes the parameters, the globals, the return variables for both f^1 and f^2; in addition, $C(f^1, f^2)$ can refer to the value of the globals at entry to a procedure using the* $\mathsf{old}(.)$ *notation.*

For an execution of a procedure f, the *summary* of the execution is a relationship between the pre and the post states of the execution. Given two programs P^1 and P^2, an indexed set of mutual summaries $C : P^1 \times P^2 \rightarrow \mathit{Formula}$, we define the *mutual summary checking* problem as follows:

> For any pair of procedures (f^1, f^2), the summaries of any pair of executions of f^1 and f^2 should satisfy $C(f^1, f^2)$.

We expect C to be sparse; i.e., it will be defined for a few pair of procedures and will be true for most pairs.

Example 1. Consider the two programs in Figure 2. Consider the following mutual summary $C(\mathsf{Foo1}, \mathsf{Foo2})$ for this pair of procedures:

$$(\mathsf{x1} = \mathsf{x2} \wedge \mathsf{old}(\mathsf{g1}) \leq \mathsf{old}(\mathsf{g2})) \implies (\mathsf{g1} \leq \mathsf{g2})$$

The summary says that if the procedures Foo1 and Foo2 are executed in a state where the respective parameters are equal, and the global g1 is less than or equal to g2 (the old(.) used to denote the state at entry to the procedures), then the resulting state (if both procedures terminate) will satisfy $\mathsf{g1} \leq \mathsf{g2}$.

Instead of specifying the preconditions, we could have alternately strengthened the antecedent in the mutual summary with $\mathsf{x1} \geq 0$. We chose to use the precondition to demonstrate the use of single program contracts in checking mutual summaries.

3.2 Checking mutual summaries

In this section, we describe a modular method for checking a program pair (P^1, P^2) annotated with a set of mutual summaries C. This consists of a method for *guaranteeing* that a mutual summary $C(f^1, f^2)$ holds and a method for *assuming* the mutual summaries of smaller executions while modularly checking the mutual summaries.

First, we define the following augmented procedure MutualCheck$\langle f^1, f^2\rangle$ that defines how to check a mutual summary $C(f^1, f^2)$:

$$\begin{array}{l}
\mathsf{void}\ \textsc{MutualCheck}\langle f^1, f^2\rangle(\mathsf{x}_1 : \mathsf{int}, \mathsf{x}_2 : \mathsf{int})\{ \\
\quad \mathsf{inline}\ \mathsf{r}_1 := \mathsf{call}\ f^1(\mathsf{x}_1); \\
\quad \mathsf{inline}\ \mathsf{r}_2 := \mathsf{call}\ f^2(\mathsf{x}_2); \\
\quad \mathsf{assert}\ C(f^1, f^2)(\mathsf{x}_1, \mathsf{x}_2, \mathsf{old}(\mathsf{g}^1), \mathsf{old}(\mathsf{g}^2), \mathsf{r}_1, \mathsf{r}_2, \mathsf{g}^1, \mathsf{g}^2); \\
\}
\end{array}$$

The only new thing to be explained is the "inline" keyword used for inlining a procedure. Consider a procedure f in a program P with a precondition ϕ, and a postcondition ψ, and body f_{body}. When we use inline at a call site of f, the call is replaced by a $\mathsf{assume}\ \phi; f_{body}$, with appropriate substitutions for the parameters of f.

Second, we define an uninterpreted summary predicate for each procedure f, and add it as a "free" postcondition for f. The "free" postconditions of a procedure are unchecked postconditions that are only assumed at call sites, but never asserted.

$$\begin{array}{l}
\mathsf{free\ post}\ R_f(\mathsf{x}, \mathsf{old}(\mathsf{g}), \mathsf{ret}, \mathsf{g}) \\
\mathsf{modifies\ g} \\
\mathsf{int}\ f(\mathsf{x} : \mathsf{int}) : \mathsf{ret};
\end{array}$$

For the cases when the global g is not modified in the procedure f, one can remove the modifies clause on g and add a postcondition $\mathsf{post\ g} = \mathsf{old}(\mathsf{g})$.

Finally, we relate the uninterpreted summary predicates of a pair of procedures using an axiom[1] (ignore the expressions inside {.} for now):

[1] An axiom in Boogie does not refer to any program state.

$$\begin{array}{l}\mathsf{axiom}(\\ \forall x_1, x_2, g_1, g_2, r_1, r_2, g_1', g_2' : \{R_{f^1}(x_1, g_1, r_1, g_1'), R_{f^2}(x_2, g_2, r_2, g_2')\}\\ \quad R_{f^1}(x_1, g_1, r_1, g_1') \wedge R_{f^2}(x_2, g_2, r_2, g_2') \implies\\ \quad C(f^1, f^2)[g_1/\mathsf{old}(g_1), g_2/\mathsf{old}(g_2), g_1'/g_1, g_2'/g_2]\\ );\end{array}$$

The axiom states that if we encounter a pair of terminating procedure calls f^1 and f^2, then we can assume the mutual summary $C(f^1, f^2)$ holds for the summaries of these two nested callees, while proving the mutual summaries of the callers. Observe that this is analogous to modular (single) program verification, where we assume the postconditions of callees while checking the contracts in the caller.

Theorem 1. *If P^1 and P^2 satisfy their single program contracts, and the contracts on each of the* MutualCheck$\langle f^1, f^2\rangle$ *procedures hold modularly for every pair of procedures $(f^1, f^2) \in P^1 \times P^2$, then the two programs satisfy the mutual summary specifications C.*

One may wonder whether the use of the quantified axiom above adds complexity to the resulting verification conditions, in particular when the contracts are expressed in a ground (quantifier-free) fragment. However, the presence of the *triggers* in the axiom (a list of expressions inside $\{.\}$, containing all the bound variables) ensure that the axioms only get instantiated at a bounded number of times. The trigger above instructs the SMT solver to instantiate this quantifier once for every pair of procedure calls of (f^1, f^2) in the verification condition, which is statically bounded.

4 Applications of mutual summaries

4.1 Equivalence checking

In various equivalence checking applications [3, 8], one checks if two procedures have identical input output behavior. For any two procedures (f^1, f^2) that are expected to be equal, the formula $C(f^1, f^2)$ can be automatically generated to be:

$$(\mathsf{x}_{f^1} = \mathsf{x}_{f^2} \wedge \mathsf{old}(\mathsf{g}_1) = \mathsf{old}(\mathsf{g}_2)) \implies \mathsf{ret}_{f^1} = \mathsf{ret}_{f^2} \wedge \mathsf{g}_1 = \mathsf{g}_2$$

There is usually an optimization for the purpose of equivalence checking: instead of using two relations for the summary of the two equivalent procedures, one can use an identical uninterpreted function for both the procedures. Using an identical uninterpreted function implicitly ensures the axiom above due to the *congruence* rule of functions, which ensures that the same input yields the same output.

On the other hand, use of an uninterpreted function restricts these analysis to deterministic procedures. The generality provided by using uninterpreted relations for a summary along with mutual summaries allows greater flexibility, without affecting the automation or efficiency needed for checking equivalence in the above cases.

```
int f1(int x1){
   if (Op[x1] == 0)
     return Val[x1];
   else if (Op[x1] == 1)
     return f1(A1[x1]) + f1(A2[x1]);
   else if (Op[x1] == 2)
     return f1(A1[x1]) - f1(A2[x1]);
   else
     return 0;
}
```

```
int f2(int x2, bool isU){
   if (Op[x2] == 0)
     return Val[x2];
   else if (Op[x2] == 1){
     if (isU)
        return  uAdd(f2(A1[x2], true),
                     f2(A2[x2], true));
     else
        return f2(A1[x2], false) +
               f2(A2[x2], false);
   }
   else if (Op[x2] == 2){
     if (isU)
       return uSub(f2(A1[x2], true),
                   f2(A2[x2], true));
     else
       return f2(A1[x2], false) -
              f2(A2[x2], false);
   }
   else
     return 0;
}
```

Fig. 3. Example for conditional equivalence.

4.2 Conditional equivalence checking

Bug fixes and feature additions result in two versions of a program that are behaviorally equivalent under a set of inputs. In *conditional equivalence* [4], the notion of equivalence can be extended to prove behavioral equivalence under a set of conditions (e.g. when the feature is turned off, or for non-buggy inputs). The key idea is to use identical uninterpreted functions for a procedure summary for inputs that would make the procedures behave the same; and use different uninterpreted functions for other inputs. We show how mutual summaries can be used for showing conditional equivalence, and more importantly helps address one of the problematic issues of using identical uninterpreted functions.

Figure 3 contains two versions of a procedure f (denoted as f1 and f2 respectively) that recursively evaluates an expression rooted at the argument x. The new version differs in functionality when an additional argument isU is provided that indicates "unsigned" arithmetic instead of the signed arithmetic represented by {+,-}.

The goal for conditional equivalence is to show that two versions of f are identical when isU is false. The following mutual summary $C(\mathsf{f1}, \mathsf{f2})$ can be used to ensure such a fact:

$$(\mathsf{x1} = \mathsf{x2} \wedge \neg \mathsf{isU}) \implies \mathsf{ret}_{\mathsf{f1}} = \mathsf{ret}_{\mathsf{f2}}$$

Earlier work [4] of using identical uninterpreted function for the two versions was problematic because of an additional parameter in the second version. Using an uninterpreted function of arity one would be unsound, as the return value of f2 depends on isU parameter. On the other hand, using an uninterpreted function of arity two would require an user to intervene to provide the second argument when modeling f1's summary.

4.3 Translation validation

Translation validation [9, 7] is a special case of equivalence checking, where the goal is to verify that a transformed program is equivalent to the original program. This is useful for verifying correct compilation of a program by a compiler. For example, a translation validator might compare a source program in a high-level language with the assembly language program generated by a compiler from the source program. Alternately, a translation validator might compare the a compiler's intermediate representation of the source program before and after each compiler phase.

This section describes how to use SymDiff to perform translation validation, focusing on the validation of compiler loop optimizations. The validation consists of three steps:

1. First, for each of the two versions of the program, loops (or, more generally, unstructured gotos) are transformed into recursive procedures, using user-provided or compiler-provided labels to guide the transformation.
2. Second, for each of the two versions of the program, calls to these recursive procedures may be inlined zero or more times to express the effect of loop optimizations such as loop unrolling.
3. Finally, mutual summaries are used to express the relation between the two versions of the program after loop extraction and inlining.

Figure 4 shows an example of three versions of a program: `Foo1` is the original program (a simple while loop), `Foo2` is an optimized version of `Foo1` (a do-while loop, with strength reduction applied to `v` and `t` hoisted from the loop), and `Foo3` is a heavily optimized version of `Foo1` (an unrolled do-while loop). The first two versions are taken from Necula [7], which describes a technique that validates the equivalence of these two versions, while the third version demonstrates loop unrolling, an optimization not handled by Necula [7]. In this example code, `g` and `n` are global constants, `a` is a global variable, and other variables are local variables. For clarity, we have replaced some of the memory references in the original example code with ordinary variable names. This simplification omits some reasoning about aliasing, but doesn't otherwise alter the nature of the example.

Loops and unstructured control SymDiff expects loops to be translated into recursive procedures. Compiler internal languages not only have loops, but may have unstructured (non-reducible) control flows built from arbitrary labels

```
int Foo1() {
  i := 0;
        //FUNCTION
  While1://LABEL
  if(i < n) {
    t := g;
    u := t * i;
    v := u + 3;
    a[i] := v;
    i := i + 1;
    goto While1;
  }
  return i;
}
```

```
int Foo2() {
  i := 0;
  if (n > 0) {
    t := g;
    v := 3;
    do2:
      a[i] := v;
      i := i + 1;
      v := v + t;
             //FUNCTION
      While2://LABEL
      if (i < n)
        goto do2;
  }
  return i;
}
```

```
int Foo3() {
  i := 0;
  v := 3;
  if (n > 1) {
    t := g;
    do3:
      a[i] := v;
      v := v + t;
      a[i + 1] := v;
      v := v + t;
      i := i + 2;
             //FUNCTION
      While3://LABEL
      if (i + 1 < n)
        goto do3;
  }
  if (i < n) {
    a[i] := v;
    i := i + 1;
  }
  return i;
}
```

Fig. 4. Translation validation example

and goto statements. To translate arbitrary unstructured control flow graphs into recursive procedures, we require the program to declare certain labels as special *function labels*. Each function label is used to generate one (possibly recursive) procedure, and each goto to that function label becomes a tail-recursive call to the procedure generated for that function label. To ensure that the resulting program has no loops, we require that every cycle in the original program pass through at least one function label.

In Figure 4, the labels `While1`, `While2`, and `While3` are marked as function labels, while the other labels (`do2` and `do3`) are ordinary local labels. In addition, the beginning of each procedure is implicitly marked with a function label. In this example, we have placed the function labels so that they match as closely as possible between the three versions of the program: in each version, the `While` function label appears directly before the loop condition. While this careful placement is not strictly necessary for performing the verification, it makes it easier to write the mutual summary formula that relates the states of the programs at the function labels.

Given a program where every cycle passes through a function label, the following simple algorithm transforms this program into a set of mutually recursive procedures. First, each function label becomes a procedure, whose parameters are all local variables and procedure parameters in scope. Second, the body for each procedure is the collection of statements reachable from that procedure's function label via paths that do not pass through a function label. Finally, each

goto statement to a function label (or implicit fall-through to a function label) becomes a tail-recursive call to the procedure for that function label. Notice that in the second step, the same statements might be included separately in different procedures, if those statements are reachable from different function labels. In the worst case, each statement could be included in each generated procedure, so the worst-case size of the resulting program is the product of the original program size and the number of function labels.

Validating optimized code Figure 5 shows the result of translating Figure 4 into recursive procedures. For example, the new procedure `Foo1` consists of only the statements that are reachable from the beginning of the original `Foo1` procedure without passing through the function label `While1`. Some statements are duplicated between procedures; for example, `do2` is reachable from both the beginning of `Foo2` and from `While2`, and thus appears in both the new `Foo2` procedure and the new `While2` procedure. To avoid confusion between immutable procedure parameters and mutable local variables, we have introduced new local variables (e.g. `i1'`) for each procedure parameter that is modified by the procedure body.

We now describe how to use SymDiff to validate the code in Figure 5, specifically the equivalence of `Foo1`, `Foo2`, and `Foo3` and the equivalence of `While1`, `While2`, and `While3`.

To validate `While1` with `While2`, we need only relate the state of these procedures as follows:

$C(\mathsf{While1}, \mathsf{While2}) =$

$(\mathsf{i1} = \mathsf{i2} \wedge \mathsf{g} = \mathsf{t2} \wedge 3 + \mathsf{g} * \mathsf{i1} = \mathsf{v2} \wedge \mathsf{old}(\mathsf{a1}) = \mathsf{old}(\mathsf{a2})) \implies (\mathsf{r1} = \mathsf{r2} \wedge \mathsf{a1} = \mathsf{a2})$

Given this, SymDiff automatically checks that `While1` with `While2` are equivalent.

In contrast to the closely related `While1` and `While2`, Figure 5's `Foo1` and `Foo2` don't look similar at first glance. Nevertheless, we can make them comparable by inlining a copy of `While1` into `Foo1`:

```
int Foo1() {
  i1 := 0;
  i1' := i1;
  if(i1' < n) {
    t1 := g;
    u1 := t1 * i1';
    v1 := u1 + 3;
    a1[i1'] := v1;
    i1' := i1' + 1;
    return While1(i1');
  }
  return i1';
}
```

```
int Foo1() {
  i1 := 0;
  return While1(i1);
}

int While1(int i1) {
  i1' := i1;
  if(i1' < n) {
    t1 := g;
    u1 := t1 * i1';
    v1 := u1 + 3;
    a1[i1'] := v1;
    i1' := i1' + 1;
    return While1(i1');
  }
  return i1';
}
```

```
int Foo2() {
  i2 := 0;
  if (n > 0) {
    t2 := g;
    v2 := 3;
    a2[i2] := v2;
    i2 := i2 + 1;
    v2 := v2 + t2;
    return While2(
      i2, t2, v2);
  }
  return i2;
}

int While2(int i2,
           int t2,
           int v2) {
  i2' := i2;
  v2' := v2;
  if (i2' < n) {
    a2[i2'] := v2';
    i2' := i2' + 1;
    v2' := v2' + t2;
    return While2(
      i2', t2, v2');
  }
  return i2';
}
```

```
int Foo3() {
  i3 := 0;
  v3 := 3;
  if (n > 1) {
    t3 := g;
    a3[i3] := v3;
    v3 := v3 + t3;
    a3[i3 + 1] := v3;
    v3 := v3 + t3;
    i3 := i3 + 2;
    return While3(
      i3, t3, v3);
  }
  if (i3 < n) {
    a3[i3] := v3;
    i3 := i3 + 1;
  }
  return i3;
}

int While3(int i3,
           int t3,
           int v3) {
  i3' := i3;
  v3' := v3;
  if (i3' + 1 < n) {
    a3[i3'] := v3';
    v3' := v3' + t3;
    a3[i3 + 1] := v3';
    v3' := v3' + t3;
    i3' := i3' + 2;
    return While3(
      i3', t3, v3');
  }
  if (i3' < n) {
    a3[i3'] := v3';
    i3' := i3' + 1;
  }
  return i3';
}
```

Fig. 5. Translation validation example

After this inlining, the following mutual summary suffices to check the equivalence of `Foo1` and `Foo2`:

$$C(\mathsf{Foo1}, \mathsf{Foo2}) = (\mathsf{old}(\mathsf{a1}) = \mathsf{old}(\mathsf{a2})) \implies (\mathsf{r1} = \mathsf{r2} \wedge \mathsf{a1} = \mathsf{a2})$$

(Currently, we rely on the user or compiler to specify the mutual summaries and to specify which procedure calls should be inlined. However, SymDiff completes the remaining equivalence checking automatically.)

The verification of `While2` to `While3` and `Foo2` to `Foo3` is similar: simply inline `While2` once into `Foo2` and inline `While2` once into itself, so that `While2` and `Foo2` express the same degree of loop unrolling that `While3` and `Foo3` express. (Note that the number of times inlining is performed will depend on the degree of loop unrolling.) Given this inlining and the following definitions, SymDiff automatically checks the equivalence of `While2` and `While3`, and `Foo2` and `Foo3`:

$C(\mathsf{Foo2}, \mathsf{Foo3}) = (\mathsf{old}(\mathsf{a2}) = \mathsf{old}(\mathsf{a3})) \implies (\mathsf{r2} = \mathsf{r3} \wedge \mathsf{a2} = \mathsf{a3})$

$C(\mathsf{While2}, \mathsf{While3}) =$

$(\mathsf{i2} = \mathsf{i3} \wedge \mathsf{t2} = \mathsf{t3} \wedge \mathsf{v2} = \mathsf{v3} \wedge \mathsf{old}(\mathsf{a2}) = \mathsf{old}(\mathsf{a3})) \implies (\mathsf{r2} = \mathsf{r3} \wedge \mathsf{a2} = \mathsf{a3})$

In summary, we can use SymDiff to check programs with loops for equivalence, even if the control flow is unstructured (non-reducible). In fact, in each case above, the SymDiff tool was able to check equivalence in just a few seconds. Nevertheless, this process is not entirely automated, since it relies on three pieces of information: the location of function labels, the desired inlining, and the definitions of mutual summaries. In the case of compiler validation, however, this information can be produced by the compiler, or, for common compiler optimizations, it might be automatically inferred, using techniques described by Necula [7].

In addition to the examples above, we have been able to encode the translation validation proofs of many common compiler optimizations using mutual summaries [5]. For example, it is possible to validate instances of software pipelining, loop peeling and restricted cases of loop fusion and loop fission. On the other hand, optimizations such as loop reversal, loop interchange that may change the order of updates to an array cannot be handled solely based on mutual summaries. We are currently investigating encoding the Permute rule [12] using mutual summaries; the rule has been needed to validate such transformations.

5 Conclusion

In this paper, we introduced mutual summaries as a mechanism for comparing programs and provided a method for checking them modularly using modern SMT solvers. Mutual summaries are general enough to encode many existing equivalence checking proofs, including those based on checking simulation relations for translation validation. More interestingly, it enables comparing programs with interprocedural changes. We are currently working on extending the technique to handle more complex interprocedural program transformations, and automating the generation of mutual summaries.

References

1. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
2. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
3. B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.
4. M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.
5. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Language Design and Implementation (PLDI '09)*, pages 327–337. ACM, 2009.
6. S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
7. G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000.
8. S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.
9. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, pages 151–166, 1998.
10. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
11. S. F. Siegel and T. K. Zirkel. Collective assertions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, LNCS 6538, pages 387–402, 2011.
12. L. D. Zuck, A. Pnueli, B. Goldberg, C. W. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.

Why3: Shepherd Your Herd of Provers*

François Bobot[1,2], Jean-Christophe Filliâtre[1,2],
Claude Marché[2,1], and Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. Why3 is the next generation of the Why software verification platform. Why3 clearly separates the purely logical specification part from generation of verification conditions for programs. This article focuses on the former part. Why3 comes with a new enhanced language of logical specification. It features a rich 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.

1 Introduction

Why3 is the next generation of the Why software verification platform. In this article, we present it as an environment for logical specification that targets a multitude of automated and interactive theorem provers. It provides a rich syntax based on first-order language and a highly configurable toolkit to convert specifications into proof obligations in various formats.

The development of Why3 is mainly motivated by the necessity to model the behavior of programs (both purely applicative and imperative) and formally prove their properties. It is commonly admitted that verification of non-trivial programs requires designing a pure logical model of the considered programs. In JML [10] or Spec# [2], for instance, such models are described using the pure fragment of the underlying programming language. In the L4.verified project [18], the Haskell language is used to model the C code of a micro-kernel. Proof assistants such as Coq [6], PVS [25], or Isabelle [17] also provide rich specification languages that are convenient to model programs.

Why3 distinguishes itself from the aforementioned approaches in that we want to provide as much automation as possible. Instead of being a theorem prover by itself, Why3 intends to provide a front-end to third-party theorem provers. To this end, we propose a common specification language which aims at maximal expressiveness without sacrificing efficiency of automated proof search

* This work was partly funded by the U3CAT (ANR-08-SEGI-021, `http://frama-c.com/u3cat/`) and DECERT (ANR-08-DEFI-005, `http://decert.gforge.inria.fr/`) projects of the French national research organization (ANR), and the Hi-lite project (`http://www.open-do.org/projects/hi-lite/`) of the System@tic ICT cluster of Paris-Région Île-de-France.

(Section 2). Another challenge is modular specification. Our proposal is a notion of reusable theories and an associated mechanism of "cloning" (Section 3). As we target a large set of theorem provers whose language and logic range from mono-sorted first-order logic to many-sorted first-order logic modulo theories to the Calculus of Inductive Constructions, we provide an extensible framework to translate the language of Why3 to these various logic languages (Section 4). Finally, we briefly describe the set of tools in the current distribution of Why3 (Section 5).

2 Logic

The logic of Why3 is a first-order logic with polymorphic types and several extensions: recursive definitions, algebraic data types and inductive predicates.

Types. A type can be non-interpreted, an alias for a type expression or an algebraic data type. For instance, the type of polymorphic binary trees is introduced as follows:

```
type tree 'a = Leaf | Node (tree 'a) 'a (tree 'a)
```

A particular case of algebraic types are enumerations.

```
type answer = Yes | No | MayBe
```

Built-in types include integers (`int`), real numbers (`real`) and polymorphic tuples. The following declaration defines the type `answer_tree` as an alias for a pair:

```
type answer_tree 'a = (tree 'a, answer)
```

Function and Predicate Symbols. Every function or predicate symbol in Why3 has a (polymorphic) type signature. For example, an abstract function that merges two integer trees can be declared as follows:

```
function merge (tree int) (tree int) : tree int
```

Both functions and predicates can be given definitions, possibly mutually recursive. As examples, we can calculate the height of a tree

```
function height (t: tree 'a) : int = match t with
  | Leaf       -> 0
  | Node l _ r -> 1 + max (height l) (height r)
end
```

or test whether the elements of a tree are both sorted and within given bounds

```
predicate sorted (t: tree int) (min: int) (max: int) =
  match t with
    | Leaf -> true
    | Node l x r ->
        sorted l min x /\ min <= x <= max /\ sorted r x max
  end
```

Why3 automatically verifies that recursive definitions are terminating. To do so, it looks for an appropriate lexicographic order of arguments that guarantees a structural descent. Currently, we only support recursion over algebraic types. Other kinds of recursively defined symbols have to be axiomatized. In future versions of Why3, we plan to allow annotating recursive definitions with termination measures. Such definitions would generate proof obligations to ensure termination.

Another extension to first-order language adopted in Why3 is inductive predicates. Such a predicate is the least relation satisfying a set of clauses. For instance, the subsequence relation over finite lists is inductively defined as follows:

```
inductive sub (list 'a) (list 'a) =
  | empty: sub (Nil: list 'a) (Nil: list 'a)
  | cons : forall x: 'a, s1 s2: list 'a.
           sub s1 s2 -> sub (Cons x s1) (Cons x s2)
  | dive : forall x: 'a, s1 s2: list 'a.
           sub s1 s2 -> sub s1 (Cons x s2)
```

Standard positivity restrictions apply to ensure the existence of a least fixed point.

Terms and Formulas. First-order language is extended, both in terms and formulas, with pattern matching, `let`-expressions, and conditional (`if-then-else`) expressions. We have decided to be faithful to the usual distinction between terms and formulas that is made in the first-order logic. Thus we make a difference between a predicate symbol and a function symbol which returns a `bool`-typed value, `bool` being defined with `type bool = True | False`. However, to facilitate writing, conditional expressions are allowed in terms, as in the following definition of absolute value:

```
function abs (x: int) : int = if x >= 0 then x else -x
```

Such a construct is directly accepted by provers not making a distinction between terms and formulas (*e.g.* provers supporting the SMT-LIB V2 format [3]). In order to translate `if-then-else` constructs to traditional first-order language, Why3 lifts them to the level of formulas and rewrites them as conjunctions of two implications.

3 Theories

Why3 input is organized as a list of *theories*. A theory is a list of *declarations*. Declarations introduce new types, functions and predicates, state axioms, lemmas and goals. These declarations can be directly written in the theory or taken from existing theories.

Figure 1 contains an example of Why3 input text, containing four theories. We start with a theory `Order` of partial order, declaring an abstract type `t` and an abstract binary predicate `(<=)`. The next theory, `List`, declares a new

```
theory Order
  type t
  predicate (<=) t t

  axiom le_refl : forall x : t. x <= x
  axiom le_asym : forall x y : t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z : t. x <= y -> y <= z -> x <= z
end

theory List
  type list 'a = Nil | Cons 'a (list 'a)

  predicate mem (x: 'a) (l: list 'a) = match l with
    | Nil -> false
    | Cons y r -> x = y \/ mem x r
  end
end

theory SortedList
  use import List
  clone import Order as O

  inductive sorted (l : list t) =
    | sorted_nil :
        sorted Nil
    | sorted_one :
        forall x:t. sorted (Cons x Nil)
    | sorted_two :
        forall x y : t, l : list t.
        x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

  lemma sorted_mem:
    forall x: t, l: list t. sorted (Cons x l) ->
    forall y: t. mem y l -> x <= y
end

theory SortedIntList
  use import int.Int
  use import List
  clone import SortedList with type O.t = int, predicate O.(<=) = (<=)

  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
end
```

Fig. 1. Example of Why3 text.

algebraic type of polymorphic lists, `list 'a`, together with a recursively defined predicate of membership.

Now we want to construct a theory `SortedList` of ordered lists. We want to reuse the definition of polymorphic lists given in theory `List`, as well as the axioms from `Order`. The `use import List` command indicates that this new theory may refer to symbols from theory `List`. These symbols are accessible in a qualified form, such as `List.list` or `List.Cons`. The `import` qualifier additionally allows us to use them without qualification. Then we `clone` theory `Order`. This is pretty much equivalent to copy-pasting every declaration from `Order` to `SortedList`. Finally, we introduce an inductive predicate `sorted` and state as a lemma that the head of a sorted list is smaller or equal to all subsequent elements.

Notice an important difference between `use` and `clone`. If we `use` a theory, say `List`, twice (directly or indirectly), there is no duplication: there is still only one type of lists and a unique pair of constructors. On the contrary, when we `clone` a theory, we create a local copy of every cloned declaration, and the newly created symbols, despite having the same names, are different from their originals.

Now, we can instantiate theory `SortedList` to any ordered type, without having to retype the definition of `sorted`. Let us build a theory `SortedIntList` for sorted lists of integers. We first import the theory of integers `int.Int` from Why3's standard library — the prefix `int` indicates the file in the standard library containing theory `Int`. The next declaration clones `SortedList` (*i.e.* copies its declarations) substituting type `int` for type `O.t` of `SortedList` and the default order on integers for predicate `O.(<=)`. Why3 controls that the result of cloning is well-typed. Notice that, when we instantiate an abstract symbol, its declaration is not copied from the theory being cloned. Thus, we do not create a second declaration of type `int` in `SortedIntList`.

Why should we clone theory `Order` in `SortedList` if we make no instantiation? Couldn't we write `use import Order as O` instead? The answer is no. When we `use` a theory, we mean to share its symbols and declarations with other places where this theory is used. On the other hand, when we `clone` a theory, we obtain a fresh, local copy of its declarations. Therefore, when cloning a theory, we can only instantiate the symbols declared locally in this theory, not the symbols imported with `use`. Therefore, we create a local copy of `Order` in `SortedList` to be able to instantiate `t` and `(<=)` later.

The mechanism of cloning bears some resemblance to modules and functors of ML-like languages. Unlike those languages, Why3 makes no distinction between modules and module signatures, modules and functors. Any Why3 theory can be `use`'d directly or instantiated in any of its abstract symbols.

4 Proof Tasks

The principal activity of Why3 can be described as processing of *proof tasks*. A proof task is basically a sequent: a list of declarations that ends with a goal. A

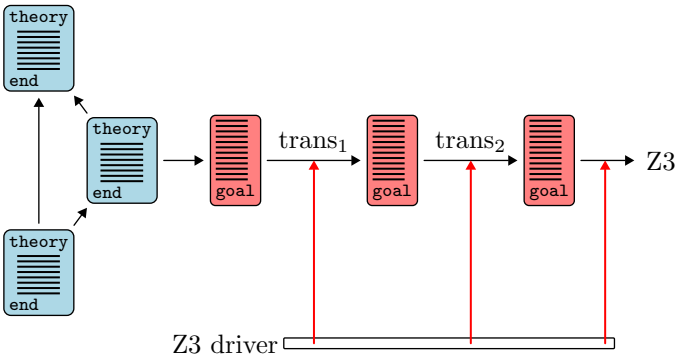


Fig. 2. Task flow in Why3.

proof task is flat: it does not contains any `use` or `clone` anymore. Why3 starts by extracting a set of proof tasks from a given theory.

Suppose we want to send a proof task to a particular prover, say Z3 [13]. Not only is the input syntax of Z3 different from Why3's syntax, there are also significant differences in the logic of the two systems. For instance, Z3 does not support polymorphism or inductive predicates. We need to apply a series of *transformations* that will gradually translate Why3's logic into the prover's logic. This series of transformations is controlled by a configuration file, called a *driver*, associated to any prover supported by Why3. The task flow from theories to provers is illustrated in Figure 2.

Figure 3 contains a simplified driver for Z3. In the driver, we specify a pretty-printer corresponding to the prover's input format (`smtv2` here). We also give regular expressions to interpret the prover output. Next we enumerate the transformations to be applied to a proof task before it can be sent to the pretty-printer. For instance, `inline_trivial` expands "simple" definitions, such as

```
predicate (>=) (x y : t) = y <= x
```

In the current implementation, we call a definition simple whenever it is non-recursive, right linear and does not contain variables at depth more than one. This might change in future versions of Why3. Transformation `eliminate_algebraic_smt` encodes algebraic data types and pattern-matching expressions in terms of uninterpreted type and function symbols [24]. Finally, `encoding_smt` eliminates polymorphic types from the proof task, converting it to an equivalent monomorphic many-sorted sequent [9]. For description of other transformations, we refer the reader to Why3's manual [8]. Finally, to take into account built-in theories of Z3, we specify the correspondence between Why3 symbols and Z3 interpreted symbols. For instance, integer addition (defined in theory `int.Int`) corresponds to the built-in operation `+` of Z3. Also, we can omit all axioms which are already known to Z3.

```
printer "smtv2"
filename "%f-%t-%g.smt"

valid "^unsat"
invalid "^sat"
unknown "^\\(unknown\\|Fail\\)" "Unknown"
time "why3cpulimit time : %s s"

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic_smt"
transformation "simplify_formula"
transformation "discriminate"
transformation "encoding_smt"

prelude "(set-logic AUFNIRA)"

theory BuiltIn
  syntax type int   "Int"
  syntax type real  "Real"
  syntax predicate  (=)  "(= %1 %2)"
end

theory int.Int
  prelude ";;; this is a prelude for Z3 integer arithmetic"

  syntax function  zero "0"
  syntax function  one  "1"
  syntax function  (+)  "(+ %1 %2)"
  syntax function  (-)  "(- %1 %2)"
  syntax function  (*)  "(* %1 %2)"
  syntax function  (-_) "(- %1)"
  syntax predicate (<=) "(<= %1 %2)"
  syntax predicate (<)  "(< %1 %2)"
  syntax predicate (>=) "(>= %1 %2)"
  syntax predicate (>)  "(> %1 %2)"

  remove prop CommutativeGroup.Comm.Comm
  remove prop CommutativeGroup.Assoc.Assoc
  remove prop CommutativeGroup.Unit_def
  remove prop CommutativeGroup.Inv_def
  (* etc. *)
end
```

Fig. 3. Driver for Z3.

Users can develop pretty-printers and transformations of their own, dynamically linked to Why3 as plug-ins. They are registered under unique names, which can be subsequently referred to in drivers. As a consequence, a user can easily add support for a new prover or tweak the interface to an existing one. For example, along with the driver in Figure 3, we provide an alternative driver for Z3 with support for built-in theory of arrays. To avoid writing the same driver rules several times, common parts can be put in separate files and included in drivers.

5 Architecture

Why3 is implemented as a OCaml programming library. Every functionality (term construction, parsing, proof task transformations, prover calls, etc.) is given in a form of an API. We took a defensive approach in designing this API: Why3 does not allow constructing an ill-formed or ill-typed term, or to use a non-declared symbol in a theory or a proof task. A special effort is made to share the common sub-terms and sub-tasks, and to memoize the intermediate results of transformations on these sub-tasks. In this way, we avoid a lot of redundant work since, in the most common case where proof tasks originate from the same theory, they share the most of their premises.

The tools we provide in Why3 distribution are built on top of this common library. We anticipate other projects to make use of this library. For instance, integrating automated theorem provers in an interactive proof assistant can be naturally done by linking with Why3 (assuming we trust the prover answers). Another way of using Why3 is to supply new parsers, transformations, or pretty-printers in the form of dynamically loadable plug-ins. As an example, we distribute a parser for the TPTP format, allowing us to test Why3 on a vast collection of theorem proving problems [28].

We package three main tools with Why3:

- a simple command-line interface `why3`, to launch a selected prover on a set of goals in a given file;
- an interactive graphical user-interface `why3ide`;
- a tool `why3bench` to benchmark different automated provers (or different configurations of the same prover) on large sets of problems. It is also useful to compare axiomatizations or transformations.

A screenshot of the Why3 GUI is shown in Figure 4. On the left side we see the available provers. We can apply some transformations to a proof task before sending it to the prover; for example, split a goal or unfold a definition in it. The goal in theory `SortedIntList` is quite simple and Alt-Ergo [7] proves it in an instant. The lemma `sorted_mem` in theory `SortedList` is, on the other hand, more difficult for automated prover since it requires induction. We thus resort to an interactive proof assistant, namely Coq, to discharge this proof task. Using the "Edit" button, the user can launch a Coq IDE to edit a proof script. After

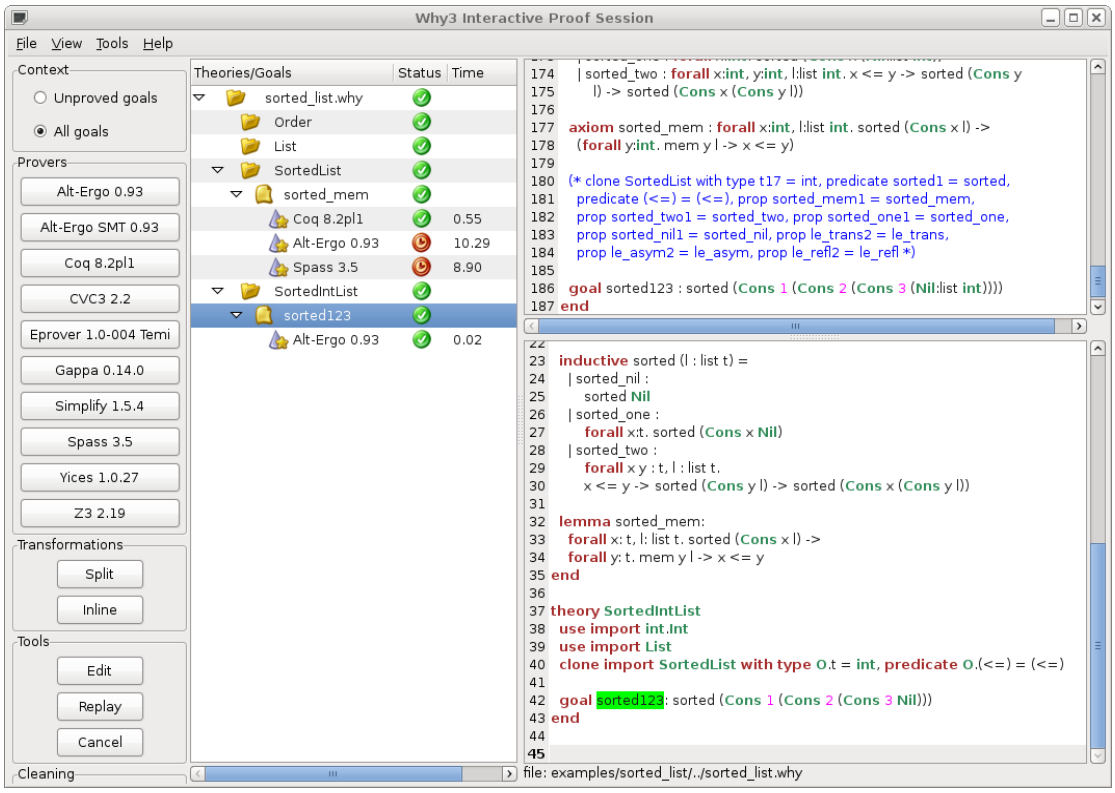


Fig. 4. Why3 GUI screenshot.

the editing session is finished, Why3 GUI rechecks that the saved proof script is accepted by Coq.

Why3 GUI saves the state of a proof session in a database file. A user can modify the initial Why3 file and then return to GUI and replay the previous proof session. For all proof tasks that have not been automatically discharged in this way, the user has to reconstruct the proof. In the future, we plan to extend Why3 GUI to a full-fledged IDE.

6 Related Work and Perspectives

We presented Why3, a language for specification and a tool to translate it into proof obligations for various interactive and automated theorem provers. Why3 improves upon the former Why 2 platform both in terms of expressiveness, architecture, extensibility (see the manual [8] for an exhaustive list of changes). The Why3 platform can be used by itself, as some kind of standalone "meta" theorem prover, but the main purpose of Why3 is to be used as an intermediate

language. For instance, we are currently designing a plug-in for the Coq proof assistant, to extract the first-order part of Coq goals and pass them to automated theorem provers. At the moment, Why3 does not attempt to analyze the output of an automated prover (neither proofs nor counterexamples). A short term perspective would be to translate counterexamples back to Why3 language. This requires not only to parse the output of a particular prover, but also to reverse the effect of various task transformations of Why3. In a long term perspective, we would like to augment the confidence in prover results by producing Coq or Isabelle certificates, in the spirit of Isabelle's Sledgehammer [22].

At the same time, we are re-implementing a verification condition generator for a programming language, WhyML, annotated with Why3 pre/post-conditions. (Interested readers are invited to visit our gallery of verified programs at `http://proval.lri.fr/gallery/why3.en.html`.) We also plan to use WhyML as an intermediate language for program verification, in the spirit of the former Why platform [14] or Boogie [1]. This is one of the two principal approaches to program verification, where a general-purpose programming language is equipped with a specification language and then possibly translated to the intermediate language of a VCGen. This is the case of VCC [12], Spec# [2], Dafny [19], Chalice [20], ESC/Java2 [11], Krakatoa [21], Frama-C [15]. The other principal approach uses a deep embedding of a programming language and its semantics in the logic of a general-purpose proof assistant. This is the case of SunRise [16], KIV [26], Isabelle/Simpl [27], KeY [5], Ynot [23], etc.

While the latter approach benefits from rich specification languages, one can argue that these languages are not close enough to the programming language constructs, creating another entry barrier for a new user. Additionally, proofs are typically performed in an interactive way, since underlying environments do not offer as much automation as state-of-the-art theorem provers (although the Sledgehammer effort strives to fill this gap). On the other hand, the former approach, which Why3 belongs to, offers more automation but poorer specification languages. We believe that Why3 features presented in this paper (inductive predicates, algebraic data types, theories) help to alleviate this drawback. We intend to promote these constructions to existing specification languages, such as ACSL [4] or JML [10].

References

1. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
4. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.
5. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
7. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.
8. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, Feb. 2011. `http://why3.lri.fr/`.
9. F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language, 2011. Preliminary report. `http://hal.inria.fr/inria-00591414/`.
10. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
11. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
12. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.
13. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`.
14. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
15. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
16. P. V. Homeier and D. F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.
17. The ISABELLE system. `http://isabelle.in.tum.de/`.
18. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Commun. ACM*, 53(6):107–115, June 2010.
19. K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.
20. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

21. C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009. `http://krakatoa.lri.fr/ws/`.
22. J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.
23. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.
24. A. Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform (version 2). Technical Report 7128, INRIA, 2010. `http://hal.inria.fr/inria-00439232/en/`.
25. The PVS system. `http://pvs.csl.sri.com/`.
26. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In W. McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.
27. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
28. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In A. Bundy, editor, *Proc. 12th Conference on Automated Deduction CADE, Nancy/France*, pages 252–266. Springer-Verlag, 1994.

coreStar: The Core of jStar

Matko Botinčan[1], Dino Distefano[2], Mike Dodds[1],
Radu Grigore[2], Daiva Naudžiūnienė[1], and Matthew J. Parkinson[3]

[1] University of Cambridge,
{matko.botincan,mike.dodds,daiva.naudziuniene}@cl.cam.ac.uk
[2] Queen Mary, University of London,
{dino.distefano,radu.grigore}@eecs.qmul.ac.uk
[3] Microsoft Research Cambridge, mattpark@microsoft.com

Abstract. Separation logic is a promising approach to program verification. However, currently there is no shared infrastructure for building verification tools. This increases the time to build and experiment with new ideas. In this paper, we outline coreStar, the verification framework underlying jStar. Our aim is to provide basic support for developing separation logic tools. This paper shows how a language can be encoded into coreStar, and gives details of how coreStar works to enable extensions.

1 Introduction

Separation logic [23] based approach to program verification has gained a lot of attention recently. It enables sound and precise reasoning about complex heap (and, generally, resource) properties which pose a major challenge to other approaches. For making separation logic an interesting hammer for program verification the crucial thing was development of tools that: (1) have automated the reasoning with separation logic [4,14,17,18,10] and (2) have scaled it to programs of substantial size [25,9,6]. These efforts were focused on automated proving of shape properties for low-level C programs, however, separation logic based verification of programs written in higher-level languages has also been approached. For instance, Distefano and Parkinson have developed jStar, a tool for verifying Java programs [15]. What is common about all these tools is that each one of them had to incorporate in one way or another a number of core techniques such as symbolic execution with separation logic [5] or loop invariants inference (e.g. like in [14]) for ensuring termination of the symbolic execution.

In general, building a program verification tool is a daunting task and requires knowledge of many domains (such as decision procedures, theorem proving, formal semantics, verification condition generation, abstract interpretation, compilation, etc.) and a complex software engineering. The line of work on the Boogie program verifier [1,20] has identified this problem and has offered a solution in terms of an intermediate verification language and a program verifier for this language. The end result has shown a lot of success and Boogie has been used as a verification backend for a number of tools including Spec# [3], VCC [11] and Dafny [19].

Motivated by the Boogie approach to program verification, this paper proposes a similar agenda for separation logic based program verification. We present coreStarIL — an intermediate language for program verification with separation logic, and coreStar — a tool enabling automated verification of coreStarIL programs. Like with Boogie, coreStarIL programs can be written manually, however, the goal is that they are automatically generated by program verifiers that take programs written in higher-level languages and encode the semantics of the input program by translating it to coreStarIL.

coreStar is the result of efforts to make the core of jStar generic and reusable. It has been successfully applied as a backend to a couple of other separation logic based program verifiers: MultiStar [24] for reasoning about multiple related abstractions with its frontend for Eiffel and VMC [8,7] for verifying multicore C programs with asynchronous memory operations. Nevertheless, many of its design choices are still open, and design decisions already been made have to be justified further. The purpose of this paper is to describe the current state of coreStar and encourage its critique and suggestions from the verification community for the further development.

2 A crash course on separation logic

Separation logic [23] helps in achieving *local reasoning* when dealing with heap allocated data structures. The idea is that verification should focus on what changes, not what stays the same. To achieve this, separation logic takes a different starting perspective than Hoare logic: instead of pre- and postconditions describing the global state, it just describes a part of the state. A precondition must describe resources that a command accesses, and everything not mentioned in the precondition is implicitly left unchanged. If the command terminates, then the resulting partial state satisfies the postcondition.

To deal with this property formally, separation logic introduces a new logical connective $*$ that expresses disjointness of partial states. The conjunction $P_1 * P_2$ says that the state can be split into two disjoint parts, one satisfying P_1 and the other P_2. This connective enables the so-called *frame rule* which allows us to extend any Hoare triple $\{P\}\, C\, \{Q\}$ by an arbitrary frame F that is unchanged by the command C, and in this way enable the local reasoning.

Any application of separation logic to automated program analysis and verification requires support of a separation logic theorem prover. The purpose of the prover is to answer queries involving separation logic formulae that occur during symbolic execution. Existing techniques for automated theorem proving in separation logic deal only with a limited fragment of separation logic (often referred to as "symbolic heaps"). In this fragment there is no negation and formulae are required to be of the form $\Delta = \Pi \wedge \Sigma$ where Π is a $\wedge$-separated sequence of pure (first-order) formulae, and Σ is a $*$-separated sequence of spatial formulae. This fragment has been proven in practice to be quite effective in terms of expressiveness as well as computational tractability.

coreStar is intended to be generic. Hence, the underlying symbolic heap representation in the tool does not hard-code any particular pure or spatial predicates (even the very basic points-to predicate $\mapsto$ is not included). A user defines his own version of separation logic to be used by providing logic rules that define how the predicates are manipulated. We provide a more detailed description about logic rules and how coreStar's separation logic theorem prover works in Sec. 5.

3 coreStar's input language coreStarIL

coreStarIL is a simple untyped imperative language that can be seen as a variant of Dijkstra's guarded commands [12]. Variants of Dijkstra's guarded commands have also been used as a basis of intermediate representation in ESC/Java [16] and Boogie [2]. The coreStarIL syntax looks as follows:

$$
\begin{aligned}
\mathit{Program} &::= \{\Delta_P\}\ (\mathit{CoreStmt};)^+\ \{\Delta_Q\} \\
\mathit{CoreStmt} &::= \bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\} \\
&\quad \mid\ \mathsf{goto}\ l_1, \ldots, l_n \\
&\quad \mid\ \mathsf{label}\ l \\
&\quad \mid\ \mathsf{abs}
\end{aligned}
$$

Here Δ_P, Δ_Q (the pre- and postcondition of the program), Pre and Post (the pre- and postcondition of commands) are symbolic heaps, $\bar{x}$ is a list of variables (we use () to denote the empty list), and $l, l_1, \ldots, l_n$ are labels. A symbolic heap is a formula of the form $\Pi \wedge \Sigma$ where Π and Σ are pure and spatial assertions, respectively, defined by:

$$
\begin{aligned}
\Pi &::= \mathsf{true} \mid E = E \mid E \neq E \mid p(\overline{E}) \mid \Pi \wedge \Pi \\
\Sigma &::= \mathsf{emp} \mid s(\overline{E}) \mid \Sigma * \Sigma
\end{aligned}
$$

(E ranges over expressions, $p(\overline{E})$ is a family of pure predicates and $s(\overline{E})$ a family of spatial predicates).

The key difference compared to the other representations [12,16,2] is use of so called *specification assignment* $\bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\}$. We explain the semantics of the specification assignment in more details in Sec. 4, but, intuitively, its purpose is to support state-modifying commands in a way that is "natural" for separation logic: the part of the state that is not touched by the command is framed away and the specification of the command is used to replace the local pre-state with the local post-state. In specification assignments we allow special variables _v that are implicitly existentially quantified (over the whole statement). The postcondition of the specification assignment can contain variables $\mathsf{ret}_1, \ldots, \mathsf{ret}_n$ that get bound to $x_1, \ldots, x_n$.

The rest of the statements are fairly standard: goto performs demonic non-deterministic jump to one of the labels denoted by label statement and abs triggers the abstraction[4] (the meaning of which is explained in Sec. 6).

3.1 coreStarIL encoding of a simple heap-manipulating language

To illustrate how to translate from a higher level language to coreStarIL we show a translation of a Smallfoot [4]-like heap-manipulating language. We consider a slightly simplified sequential version of Smallfoot with the commands of the following syntax:

$$
\begin{array}{lll}
A ::= & \mathsf{empty} & \text{empty command} \\
\mid & x := E & \text{variable assignment} \\
\mid & x := [E] & \text{heap lookup} \\
\mid & [E] := F & \text{heap mutation} \\
\mid & \mathsf{new}(x) & \text{allocation} \\
\mid & \mathsf{dispose}(x) & \text{deallocation} \\
C ::= & A \mid C; C \mid x := f(E) & \\
\mid & \mathsf{if}(E)\,\{C\}\,\mathsf{else}\,\{C\} \mid \mathsf{while}(E)\,\{C\} &
\end{array}
$$

Atomic commands. Since atomic commands A allow reading and mutation of heap we have to somehow represent the heap contents. The standard way in separation logic is to use the spatial predicate $\mapsto$ such that $x \mapsto v$ iff the local heap contains a single cell x that points to the value v. Following the semantics of the atomic commands [5] we translate them to coreStarIL statements as follows:

$$
\begin{array}{rcl}
tr(\mathsf{empty}) & = & () := \{\}\{\}; \\
tr(x := E) & = & x := \{\}\{\mathsf{ret} = E\}; \\
tr(x := [E]) & = & x := \{E \mapsto \_\mathsf{v}\}\{E \mapsto \_\mathsf{v} \wedge \mathsf{ret} = \_\mathsf{v}\}; \\
tr([E] := F) & = & () := \{E \mapsto \_\mathsf{v}\}\{E \mapsto F\}; \\
tr(\mathsf{new}(x)) & = & () := \{\}\{x \mapsto \_\mathsf{v}\}; \\
tr(\mathsf{dispose}(x)) & = & () := \{x \mapsto \_\mathsf{v}\}\{\};
\end{array}
$$

In the translated statements _v is a fresh unifiable variable. For readability we have omitted explicitly denoting the translation of expressions.

If-else and while. coreStarIL does not contain assert and assume statements like languages in [12,16,2] as these statements can in fact be encoded using specification assignment:

$$
\begin{array}{rcl}
\mathsf{assert}(E) & \triangleq & () := \{E\}\{\} \\
\mathsf{assume}(E) & \triangleq & () := \{\}\{E\}
\end{array}
$$

[4] Although in principle we can perform abstraction automatically at dominators of basic blocks, we want to allow frontend to decide when to perform the abstraction.

Using assume statements the if-else and while commands can be translated in a standard way:

$$\begin{array}{rcl} tr(\mathsf{if}(E)\,\{C_1\}\,\mathsf{else}\,\{C_2\}) & = & \mathsf{goto}\ l_1, l_2; \\ & & \mathsf{label}\ l_1;\ \mathsf{assume}(E);\ tr(C_1)\ \mathsf{goto}\ l_3; \\ & & \mathsf{label}\ l_2;\ \mathsf{assume}(\neg E);\ tr(C_2)\ \mathsf{goto}\ l_3; \\ & & \mathsf{label}\ l_3; \\ tr(\mathsf{while}(E)\,\{C\}) & = & \mathsf{label}\ l_1;\ \mathsf{abs};\ \mathsf{goto}\ l_2, l_3; \\ & & \mathsf{label}\ l_2;\ \mathsf{assume}(E);\ tr(C)\ \mathsf{goto}\ l_1; \\ & & \mathsf{label}\ l_3;\ \mathsf{assume}(\neg E); \end{array}$$

where l_1, l_2 and l_3 are fresh labels. Note that we are putting the abs statement at the head of the loop to allow abstraction (though a translation equivalent to what the Smallfoot tool does would not contain it).

Procedure calls. A procedure call is translated by using the callee's specification and encoding it into the specification assignment. If $\lambda y\,.\,\{P\}f(y)\{Q\}$ is the specification of a function f then

$$tr(x := f(E)) \quad = \quad x := \{P[E/y]\}\{Q[E/y]\}$$

In order to avoid capture, the variables in the specification of f need to be freshened before the substitution takes place.

4 Symbolic execution in coreStar

In coreStar, the symbolic execution is performed on the control-flow graph of the input program. The symbolic states are defined as pairs of a CFG node and a symbolic heap. Statements are executed symbolically over the symbolic states, i.e., at each step the current symbolic state is updated to reflect the abstract effect of the statement. The process executes until it reaches a fix-point where no new symbolic states can be reached.

The input to the symbolic execution is a coreStarIL program of the form $\{\Delta_P\}s_1;\ldots s_n;\{\Delta_Q\}$. coreStar provides two kinds of symbolic execution:

- symbolic execution with frame inference;
- bi-abductive symbolic execution.

In the classical symbolic execution with frame inference, assuming Δ_P statements are symbolically executed aiming to reach at the end a symbolic state implying Δ_Q. However, this approach assumes that we have given a sufficient precondition for $s_1;\ldots s_n;$. Sometimes we want to allow incomplete preconditions (e.g., because we want to relieve the user of the burden to write down the specification or because we may just not know the complete specification). coreStar can try to find out the missing part of the precondition Δ_M such that $\{\Delta_P * \Delta_M\}s_1;\ldots;s_n\{\Delta_Q\}$ (even if Δ_P is empty). This approach requires using a technique called bi-abduction [9] so we are referring to it as bi-abductive symbolic execution.

4.1 Symbolic execution with frame inference

Classical symbolic execution for separation logic requires frame inference, that is: given Δ_1 and Δ_2 find Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds. The frame inference allows us to execute a command C with a precondition P and a postcondition Q from a current state Δ_H in the following way:

$$\frac{\{P\}\,C\,\{Q\} \qquad \Delta_H \vdash P * \Delta_F}{\{\Delta_H\}\,C\,\{Q * \Delta_F\}}$$

If there exists Δ_F such that $\Delta_H \vdash P * \Delta_F$ then this rule says that from a pre-state Δ_H we can propagate $Q * \Delta_F$ to the post-state.

4.2 Bi-abduction

Bi-abduction [9] can be seen as a generalisation of the frame inference: given Δ_1 and Δ_2 the goal is to find the "missing" assumption (anti-frame) Δ_A and the frame Δ_F such that $\Delta_1 * \Delta_A \vdash \Delta_2 * \Delta_F$ holds. The authors of [9] have considered solving the bi-abduction query by utilizing separate proof systems for frame inference and abduction. In coreStar we solve the bi-abduction query using a single proof system as it will be explained in Sec. 5.

Using the Hoare's rule of consequence, the procedure for bi-abduction gives rise to a bi-abductive version of the frame rule:

$$\frac{\{P\}\,C\,\{Q\} \qquad \Delta_H * \Delta_A \vdash P * \Delta_F}{\{\Delta_H * \Delta_A\}\,C\,\{Q * \Delta_F\}}$$

The rule tells us that given a pre-state Δ_H and a specification of a command $\{P\}\,C\,\{Q\}$ we can compute Δ_F and Δ_A such that $\Delta_H * \Delta_A$ is sufficient to execute C in order to obtain the post-state $Q * \Delta_F$.

4.3 Bi-abductive symbolic execution

Now we describe how the symbolic execution with bi-abduction operates in coreStar. To simplify the presentation we represent the input program by a set of nodes N, and the functions $succ : N \to N$ returning the successor node which represents the statement that follows in the input program and $stm : N \to \mathcal{S}$ returning the statement associated with the node ($\mathcal{S}$ denoting the set of coreStarIL statements). Symbolic execution operates by traversing pairs of the form $(n, (\Delta_H, \Delta_M))$ where $n \in N$ is a node, Δ_H is the symbolic heap representing the current state and Δ_M is the missing part of the heap discovered so far which needs to be added to the precondition.

Fig. 1 shows the symbolic execution rules that define the effect of each coreStarIL statement on a symbolic state. If $(n, (\Delta_H, \Delta_M)) \overset{s}{\rightsquigarrow} (n', (\Delta'_H, \Delta'_M))$ then executing s in the state $(n, (\Delta_H, \Delta_M))$ leads to the new state $(n', (\Delta'_H, \Delta'_M))$. All rules apart from the rule for specification assignment are straightforward. To execute $\bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\}$ in a pre-state (Δ_H, Δ_A) we proceed as prescribed

$$
\begin{array}{lll}
(n,(\Delta_H,\Delta_M)) & \overset{\bar{x}:=\{\mathsf{Pre}\}\{\mathsf{Post}\}}{\rightsquigarrow} & (succ(n),(\Delta_F * \mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}],\Delta_A * \Delta_M)) \text{ where} \\
 & & \Delta_H * \Delta_A \vdash \mathsf{Pre} * \Delta_F \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{goto}\ l_1,\ldots,l_n}{\rightsquigarrow} & (n_{l_i},(\Delta_H,\Delta_M)) \text{ where } stm(n_{l_i}) = \mathsf{label}\ l_i,\ i = 1..n \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{label}\ l}{\rightsquigarrow} & (succ(n),(\Delta_H,\Delta_M)) \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{abs}}{\rightsquigarrow} & (succ(n),abs(\Delta_H,\Delta_M))
\end{array}
$$

Fig. 1: Symbolic execution rules for coreStarIL statements.

by the bi-abductive version of the frame rule. We first invoke the prover to provide us with the pair of symbolic heaps (Δ_F, Δ_A) solving the associated bi-abduction query, and then conjoin (Δ_F, Δ_A) with $(\mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}], \Delta_M)$ to obtain $(\Delta_F * \mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}], \Delta_A * \Delta_M)$ as a post-state.

The first and the second rule in Fig. 1 are nondeterministic (the nondeterminism in the first rule may arise due to multiple answers to the frame inference question, and in the second due to multiple labels) thus the transition to a new symbolic state may introduce more than just a single symbolic heap pair associated with the destination node. Thus in coreStar the nodes are associated with sets of symbolic heap pairs and are traversed by employing a worklist algorithm. Each such set of symbolic heaps pairs associated to a node represents a disjunction over its elements.

After the symbolic execution completes we are left with a missing part in the anti-frame that together with the starting Δ_P forms a new candidate precondition $\Delta_{P'}$. To check that the $\Delta_{P'}$ is sufficient to execute the program we re-run the symbolic execution starting with precondition $\Delta_{P'}$, this time with the frame inference only.

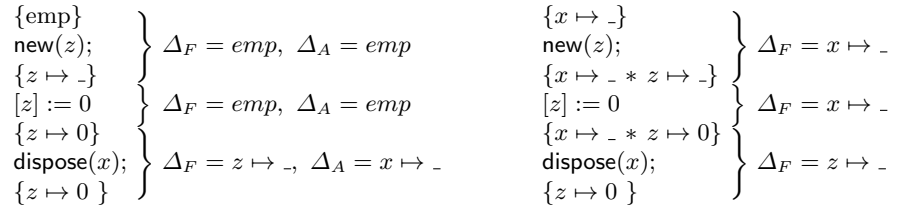
Example. To illustrate how the bi-abductive symbolic execution works let us consider the example in Fig. 2 (a). Symbolic execution with frame inference only would fail at the third statement program since we do not have a heap cell for x. Bi-abductive symbolic execution at this step finds the missing part $\Delta_A = x \mapsto \_$. After forming a new candidate precondition we run the symbolic execution with frame inference as shown in Fig. 2 (b).

5 The coreStar's separation logic theorem prover

As we have seen in Sec. 4, the purpose of the separation logic theorem prover in coreStar is to provide an answer to three types of queries:

- the frame inference;
- the bi-abductive frame inference; and
- deciding implications between symbolic heaps.

The last type of query occurs when checking whether the final disjunctive state in the symbolic execution satisfies the desired postcondition. Answering all three types of queries reduces to a particular proof search task.



(a) Bi-abductive frame inference (b) Frame inference

Fig. 2: Symbolic execution example

5.1 Prover's internals

The design of the coreStar's separation logic prover builds on the design of the entailment prover in Smallfoot [5] and descendent tools. In contrast to these prior tools, the coreStar's prover works with an internal representation that allows performing the bi-abductive frame inference directly. We first explain how the prover performs the frame inference, and then show what is different for the other two types of queries.

Frame inference. To perform frame inference coreStar's prover works with sequents $\Sigma \mid \Delta_A \vdash \Delta_G$, where Δ_A is the assumed formula, Δ_G is the goal formula and Σ is the subtracted (spatial) formula. The semantics of this judgment is $\Sigma * \Delta_A \;\Rightarrow\; \Sigma * \Delta_G$.

As in Smallfoot, in order to answer the frame inference question, starting with a sequent $\mathsf{emp} \mid \Delta_1 \vdash \Delta_2$ the prover searches for sequents of the form $\Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp}$. All such leftover formulae Δ_L are collected from the leaves of the proof search tree and their disjunction forms the frame that was searched for. Namely, an incomplete proof

$$\begin{array}{c} \Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp} \\ \vdots \\ \mathsf{emp} \mid \Delta_1 \vdash \Delta_2 \end{array}$$

can be transformed into the desired proof of $\Delta_1 \vdash \Delta_2 * \Delta_L$ by spatially adding Δ_L to the right hand side at every proof step, and in addition, allowing $*$-introduction of Δ_L at the top of the proof.

Bi-abductive frame inference. The proof search in bi-abductive frame inference is performed using sequents $\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M$, where Δ_A is the assumed formula, Δ_G is the goal formula, Σ is the subtracted (spatial) formula and Δ_M represents the anti-frame part of the formula. The sequents that the prover searches for in this case are of the form $\Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp} \dashv \Delta_M$. The frame and the anti-frame are formed from a disjunction of all leftover pairs of formulae Δ_L and Δ_M.

Semantics of our bi-abductive proof sequents can be described in the following way. Assume that the sequent $\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M$ appears in the proof search that has started from a symbolic execution pre-state (Δ_H, Δ_P). Then the semantics of the sequent is given by

$$\Sigma * \Delta_A * \Delta_M * \Delta_P \Rightarrow \Sigma * \Delta_G * \Delta_H.$$

In other words, one can see the symbolic execution with bi-abductive frame inference as performing a proof search for the given sequence of statements using the sequents of the form

$$\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M \dashv\vdash \Delta_H, \Delta_P$$

with the above semantics.

Deciding implication. To decide the implication the prover works like in the frame inference case but searches for sequents of the form $\Sigma \mid \mathsf{emp} \vdash \mathsf{true} \wedge \mathsf{emp}$.

5.2 Logic rules

The prover engine of coreStar is designed to be agnostic to specific details of the underlying logical theory. Except for basic constants true and emp, equalities, disequalities and the multiplicative connective, no other pure or spatial predicates are predefined in coreStar. Support for a particular version of separation logic is provided by specifying rewrite and proof rules in input files. This allows coreStar to deal not just with heap-specific reasoning (like most tools for shape analysis with separation logic do) but to reason in general about any objects that can be represented by means of abstract predicates [21] like e.g., threads, locks, abstract datatypes, etc.

Proof rules. Some general structural rules that need to be used with any kind of separation logic proof system are hard-coded into the prover. However, the rules that define how the reasoning within a particular logic theory is performed are specified externally and have to be provided by the user.[5] coreStar loads the proof rules into the separation logic prover and during the proof search applies the rules in order as they are specified in the input file. coreStar proof rules are specified using the syntactic construct `rule` [13]:

```
rule X: | Concl-L |- Concl-R -| Concl-A
if SubtFPre | Prem-L |- Prem-R -| Prem-A
```

The keyword `if` separates the conclusion from the premise. Mathematically this syntax corresponds to

$$\frac{\Sigma * \texttt{SubtFPre} \mid \Delta_A * \texttt{Prem-L} \vdash \Delta_G * \texttt{Prem-R} \dashv \Delta_M * \texttt{Prem-A}}{\Sigma \mid \Delta_A * \texttt{Concl-L} \vdash \Delta_G * \texttt{Concl-R} \dashv \Delta_M * \texttt{Concl-A}}\ \mathrm{X},$$

for some Σ, Δ_A, Δ_G and Δ_M. Notice that this means that proof rules are implicitly framed – everything that is not mentioned in the definition of the proof rule is left unchanged.

[5] At the moment of writing this paper coreStar does not check that user-defined rules are consistent. Therefore, the user should ensure the soundness of rules.

```
rule pto_match:  | x ↦ y  |- x ↦ t
without  y ≠ t
if  x ↦ y  |  |- y = t
or  x ↦ y  |  y = t  |- -|  y = t
```

```
rule pto_missing:  |  |- x ↦ y
if  |  |- -|  x ↦ y
```

Fig. 3: Proof rules for the points-to predicate.

Example. The coreStar proof rules for bi-abductive reasoning with standard separation logic points-to predicate are shown in Fig.3. The empty parts of the sequents stand for empty heap emp (and therefore are simply preserved), and the question mark is used to denote variables that can be unified with any expression.

The rule `pto_match` says if we have a heap cell with the same address in both the assumed and the goal formula of an entailment, then move the cell to the matched (subtracted) formula, and either add a proof obligation that the corresponding values are the same to the goal formula, or abduct this equality and add it to the assumption and the anti-frame. The `without` clause prevents firing of the rule if we already know the cells have different values (this way we can avoid applying the rule infinitely). The rule `pto_missing` performs abduction of a heap cell and adds it to the anti-frame in case the rule `pto_match` did not fire and we still have the heap cell in the obligation.

Rewrite rules. The prover also can be extended with rewrite rules which are used to simplify terms. For example, the following rewrite rules would be used to simplify list terms:

`rewrite cons_hd:` $hd(cons(x, y)) = x$
`rewrite cons_tl:` $tl(cons(x, y)) = y$

6 Abstraction

Symbolic execution alone does not converge in many cases. To ensure termination, symbolic states need to be abstracted. In coreStar, abstraction is invoked on abs statements. Abstraction is critical for the success of the symbolic execution. If abstraction forgets too much information then next steps of symbolic execution may fail due to too weak symbolic state. If abstraction keeps too much information then the computation may never converge.

6.1 Abstraction rules

Abstraction rules help the convergence of coreStar's computation by syntactic rewriting of predicates. The idea is that an abstraction rule simplifies a formula so that it remains within a restricted class of heaps for which simplification is known to converge. Abstraction rules are of the form:

$$\frac{\text{condition}}{\Delta_H * \Delta'_H \rightsquigarrow \Delta_H * \Delta''_H} \text{ (Abs Rule)}$$

Heap Δ'_H gets replaced by Δ''_H if the condition holds. Δ''_H should be more abstract (simpler) than Δ'_H since some unnecessary information is removed (abstracted away). Heap Δ_H is an arbitrary context preserved by the abstraction rule.

Given a set of abstraction rules, coreStar tries to use any rules that can be applied to a heap. When no rules are applicable the resulting heap is maximally abstracted. Note that to ensure termination of this strategy, abstraction rules should be chosen so that each application strictly simplifies the heap. Moreover, for soundness, the abstraction rules *must* be true implications in separation logic. When designing abstraction rules checking this implication gives an easy sanity condition for the soundness of the resulting fixed-point computation.

Example. The abstraction rules used for linked lists are based on how programmers typically deal with linked list programs. For example, suppose we had a predicate $\mathsf{node}(x, y)$ representing a node at address x with the next pointer y, and a predicate $\mathsf{lseg}(x, y)$ representing a linked list starting at y and ending with a pointer to y. We might want to have the following rules for abstraction [14]:

$$\begin{aligned}
\exists x' \,.\, \mathsf{node}(x, x') \ast \mathsf{node}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil}) \\
\exists x' \,.\, \mathsf{lseg}(x, x') \ast \mathsf{node}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil}) \\
\exists x' \,.\, \mathsf{lseg}(x, x') \ast \mathsf{lseg}(x', \mathsf{nil}) &\rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})
\end{aligned}$$

In coreStar, the last rule, for instance, would be specified as

$$\frac{\_x \notin \mathit{Context} \cup \{x\}}{\mathsf{lseg}(x, \_x) \ast \mathsf{lseg}(\_x, \mathsf{nil}) \rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})}$$

The heap Δ_H is implicitly added to both sides of $\rightsquigarrow$. The condition in the rule says that $\_x$ does not occur syntactically in the rest of the heap (i.e., $\_x \notin \mathit{Var}(\Delta_H)$) and that x cannot be instantiated to $\_x$.

7 Conclusions and future work

The tool coreStar retains the Java-agnostic parts of the program verifier jStar — a prover and a symbolic interpreter. The prover answers three types of queries: Given Δ_1 and Δ_2, (1) find Δ_A and Δ_F such that $\Delta_1 \ast \Delta_A \vdash \Delta_2 \ast \Delta_F$ holds, (2) find Δ_F such that $\Delta_1 \vdash \Delta_2 \ast \Delta_F$ holds, and (3) decide $\Delta_1 \vdash \Delta_2$. Here, Δs are separation logic formulas. A few symbols, such as equality and emp, are interpreted by the prover directly; a few other symbols, such as addition, are interpreted by an SMT solver, which is asked to reason about the pure parts of the formulas; all other symbols, including predicates like points-to, are interpreted according to a set of logical rules provided by the user.

The symbolic interpreter works on programs written in coreStarIL, which is a very simple language. Its control flow is unstructured and the only interesting statement is the specification assignment. The interpreter (tries to) answer two types of questions: Given a program P with precondition Δ_1 and postcondition Δ_2, (4) find Δ_A such that $\{\Delta_1 \ast \Delta_A\}\, P\, \{\Delta_2\}$ is a valid Hoare triple

and (5) decide $\{\Delta_1\}\, P\, \{\Delta_2\}$. In both cases, it over-approximates the possible executions according to a set of user-provided abstraction rules.

The interface to the prover (tasks 1–3 above) is a fairly stable OCaml module signature; the interface to the symbolic interpreter (tasks 4 and 5 above) is coreStarIL. We believe that many tools based on separation logic contain implementations of special cases of tasks 1–5. For example, Sec. 3.1 essentially shows how the Smallfoot tool [4] could be reimplemented as a small frontend for coreStar.

Much of the burden of ensuring soundness, however, remains with the frontend, because coreStar believes, without checking, that all the (logical and abstraction) rules it is given are sound. We are exploring several approaches to make coreStar less gullible. One option is to formalise coreStar's proof system in a higher-order prover and accept only rules that come with a soundness proof. Another option is to design a higher level language for rules that curtails expressivity such that the soundness of logical rules is decidable. We could also restrict abstraction rules so that their confluence and termination is decidable.

On the other hand, there are situations where we feel that the current abstraction rules are not flexible enough, which is why coreStar has a prototype mechanism to allow plugging in arbitrary abstract domains. When such plugins are in use, how much coreStar trusts its user is not the only issue — now it becomes important how much the user should trust coreStar. To tackle this problem, we are considering generating proofs (and requiring plugins to provide proofs) that can be checked easily.

Currently, coreStar provides no support for verifying large programs. For example, in jStar it is the frontend's responsibility to encode method calls as coreStarIL specification assignments. Once coreStarIL has procedures and procedure calls, coreStar should implement several global analyses, such as specification inference for mutually recursive procedures via bi-abduction [9] or RHS [22].

References

1. 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.
2. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS*, 2004.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
6. Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.
7. Matko Botinčan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Proving the safety of asynchronous memory operations in multicore programs. *Submitted*.

8. Matko Botinčan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Automatic safety proofs for asynchronous memory operations. In *PPOPP*, pages 313–314, 2011.
9. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
10. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *ICECCS*, pages 307–320, 2007.
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. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
13. Dino Distefano, Mike Dodds, and Matthew J. Parkinson. How to verify java program with jStar: a tutorial. `http://www.jstarverifier.org/jstar.tutorial.pdf`.
14. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
15. Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
16. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.
17. Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, pages 240–260, 2006.
18. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *APLAS*, pages 304–311, 2010.
19. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
20. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.
21. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
22. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
23. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
24. Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with multistar. In *OOPSLA*, pages 504–519, 2010.
25. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

Corral: A Whole-Program Analyzer for Boogie

Akash Lal, Shaz Qadeer, and Shuvendu Lahiri

Microsoft Research
{akashl, qadeer, shuvendu}@microsoft.com

Abstract. This paper presents Corral, a whole-program analyzer for Boogie programs. Corral looks for a feasible execution of the program that leads to an assertion failure. The execution may span multiple procedures and may iterate loops multiple times. Unlike the Boogie verifier, Corral does not require any user annotations.

1 Introduction

Boogie [5] is an expressive language that serves as the intermediate representation of many static-analysis tools. Boogie comes with a modular program verifier. It expects user annotations in the form of procedure pre/post conditions and loop invariants. In return, it can either report that the program is correct, or that some user annotation or program assertion is possibly incorrect. The user annotations allow the Boogie verifier to be scalable because it only needs to look at one procedure at a time. Consequently, the verifier forms an important part of source-level annotation-based tools like Havoc [11] and Spec# [4].

This paper presents Corral, a new verifier for Boogie programs. Corral is a whole-program analysis tool. It does not require user annotations. Given a program with a distinguished `main` procedure (from which program execution starts) and a set of assertions inside the program, Corral looks for a program execution that can violate an assertion. Corral is precise for bug-finding: (1) Given a program with a bug, Corral will eventually find it; and (2) if Corral reports a bug, it is indeed a true bug. That is, Corral does not report false positives. Corral can also prove correctness of certain programs, however, there are correct programs for which Corral will not terminate[1]. Thus, in practice, we bound Corral's search as explained later.

Being a whole-program analyzer, we believe that Corral provides a good alternative to Boogie's default verifier and will enable Boogie to be used in many more applications. Using Corral, we have been able to build Poirot [16], a bug-finding tool for concurrent C and .NET programs and have used it to find real bugs in Windows device drivers.

Corral's search algorithm is not based on invariant discovery, unlike many of the existing software-verification tools [2, 9, 7]. There are two reasons for this choice. First, invariant discovery in the presence of arrays and arithmetic, which form an integral part of Boogie, is challenging. Second, an important source of

[1] After all, the problem addressed by Corral is undecidable.

programs for Corral are sequential programs that are produced from concurrent programs after imposing a bound on the number of context-switches [6, 14, 12]. Such sequential programs often have very complicated invariants because they simulate concurrency using data nondeterminism and invariant discovery for such programs can be difficult. SLAM [2] did not work well when we tried it on such machine-generated programs, whereas, as our experiments show (§4), Corral is seamlessly able to handle them.

Corral is based on the idea of *recursion bounding*: if one imposes a bound on the number of loop iterations and recursive calls in the program, then the resulting problem of assertion checking is decidable, and one that can be checked using a theorem prover. Consequently, Corral has an iterative recursion-bounding loop. First, it converts all loops to recursive procedures. Next, it starts with the recursion bound r initialized to 1. Then it uses the theorem prover to explore all program behaviors in which the amount of recursion is bounded by r. If a bug is found, it is reported to the user (it must be a true bug). If no bugs are found, then the bound r is incremented and the process is repeated until a timeout is reached or a bug is found.

There is a simple way of implementing the above loop in Boogie using procedure inlining. Once the recursion bound is fixed to r, one can statically inline all procedures up to depth r to construct a single-procedure program without loops. Then the Boogie verifier is capable of looking for assertion failures in this program. The problem with this approach is that static inlining can result in an exponential blow-up in program size. In our experience, even for medium-sized programs with a small recursion bound, static inlining can quickly run out of memory either in creating the inlined program or in constructing its verification condition. This is unfortunate because assertions in the program may have been easily proved or disproved if the theorem prover got a chance to look at the verification condition. We also tried using quantifiers and mutually-defined predicates to describe procedures, but the theorem prover would often get lost trying to instantiate quantifiers without much guidance.

For the reasons mentioned above, Corral uses a novel goal-directed search that we call *stratified inlining*. Instead of inlining all procedures up to a given bound, it only inlines a few. If these are sufficient to find an assertion violation, then we report it to the user. If an assertion violation is not found, we change the verification condition so that the theorem prover tells us which procedures to inline next. This algorithm is explained in more detail in §2. The algorithm also applies a simple abstraction that allows it to prove correctness (irrespective of the recursion bound) in some cases.

Our vision for Corral is that all of the semantic exploration should be done inside the theorem prover, whereas intuitions about programs and typical bugs should be used outside it. The stratified inlining algorithm follows this vision. It is based on the intuition that often not all procedures are required to find bugs and one can quickly rule out large parts of the program as being "safe". Our next intuition is that not all program variables are necessary to reveal bugs. Consequently, Corral uses *variable abstraction* on global variables.

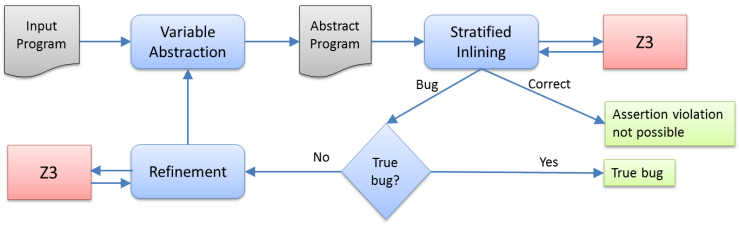


Fig. 1. CORRAL's architecture.

Let G be the set of all global variables of a program. CORRAL maintains a set of *tracked variables* $T \subseteq G$ and abstracts away all global variables in $G - T$. Next, it uses a novel counterexample-guided refinement procedure to enlarge the set T if the current abstraction is too coarse. Variable abstraction is a coarse-grained abstraction when compared to predicate abstraction. Its advantage is its simplicity (it works for programs with arrays without change) and the fact that refinement cannot *diverge* (i.e., one can have at most $|G|$ number of refinements and no more).

The work flow of CORRAL is shown in Fig. 1. It uses Z3 [18] as the theorem prover. CORRAL starts with an empty set of tracked variables T. It abstracts the input program using T. Next, it feeds the abstracted program to stratified inlining to look for assertion violations. If it find one, then we check the path against the entire set of globals variables. If the path is feasible, then it is a true bug. Otherwise, we call our refinement routine to increase the set of tracked variables, and then repeat the process. Note that, besides the outer loop that increases T, both the stratified inlining (§2) and refinement (§3.1) boxes have their own iterative loops. They make multiple calls to Z3, denoted by the two-way arrows to the Z3 box in Fig. 1.

The architecture of CORRAL is similar to that of STORM [12]. However, STORM uses static inlining and a simple refinement algorithm. We show (in §4) that stratified inlining performs much better than static inlining, and CORRAL's refinement algorithm performs much better than STORM's algorithm.

The rest of the paper talks about the stratified inlining algorithm (§2), the abstraction-refinement algorithm (§3) and experiments (§4). Proofs of theorems and Lemmas can be found in our technical report [13].

2 Stratified Inlining

The goal of stratified inlining is to take a Boogie program (possibly with recursion) as input and find a feasible trace ending in an assertion violation, if there is one. We describe the algorithm on a simple imperative programming language. A program in our language has a single global variable g and a collection of procedures, each of which have a single input parameter i and a single output parameter o. (We will show later how to lift these restrictions.) The body of a

procedure comprises of a sequence $\bar{l}$ of local variable declarations followed by a statement S:

$$S ::= \mathit{havoc}\ x \mid x := e \mid \mathit{assume}\ e \mid \mathit{call}\ x := \mathtt{p}(e) \mid S; S \mid S \Box S$$

Note that we do not have any loops in our simple programming language; we assume that all loops have been converted into recursive procedures. Other control flow is modeled using the sequence and choice operators, denoted by ; and $\Box$ respectively. The commands *havoc* x, $x := e$, and *call* $x := \mathtt{p}(e)$ modify the variable x, which must either be the global variable g, a local variable of the enclosing procedure, or the output parameter o; the input parameter i cannot be modified.

Given a program P, a procedure `m` in the program, an initial condition $\varphi(i, g)$ over i and g, and a final condition $\psi(o, g)$ over o and g, the goal is to determine whether there is an execution of `m` that begins in a state satisfying φ and ends in a state satisfying ψ. We begin with an informal description of our algorithm, shown in Fig. 2; a more formal treatment follows later.

Input: An initial condition φ and a final condition ψ
Input: The starting procedure `m`.
Output: CORRECT or BUG(τ)
1: P := {assume φ; `m`; assert $\neg\psi$}
2: C := *open-call-sites*(P)
3: **for** $r = 1$ **to** MAX **do**
4: **while** *true* **do**
5: *//Query-type 1*
6: $P' = P[\forall_{c \in C}\ c \leftarrow$ assume false$]$
7: **if** *check*(P') == BUG(τ) **then**
8: **return** BUG(τ)
9: *//Query-type 2*
10: C_1, C_2 := *split-on-cost*(C, r)
11: $P' = P[\forall_{c \in C_1} c \leftarrow \mathit{summary}(c), \forall_{c \in C_2}\ c \leftarrow$ assume false$]$
12: ret := *check*(P')
13: **if** ret == CORRECT $\wedge$ $C_2 == \emptyset$ **then**
14: **return** CORRECT
15: **let** BUG(τ) = ret
16: $P := P + \{\mathit{inline}(c) \mid c \in C, c \in \tau\}$
17: C := *open-call-sites*(P)
18: **if** $C_1 == \emptyset$ **then**
19: **break**

Fig. 2. An informal description of the stratified inlining algorithm.

Stratified inlining uses under- and over-approximation of procedure behaviors to inline calls on demand. To underapproximate a procedure, we simply block all executions through it, i.e., we replace it with "assume false". To overapproximate a procedure, we let it modify the global variable g arbitrarily and return an arbitrary value for the output parameter o. We call this overapproximation of a procedure `p` as *summary*(`p`).[2]

The algorithm maintains a partially-inlined program P starting from the procedure `m` (line 1). It also maintains the set of non-inlined (or *open*) calls in P in the variable C. The loop that follows is the recursion bounding loop. Each iteration of the loop searches for a bug (a program path) with r recursive calls or fewer.

Each iteration of the loop at line 4 comprises of two main steps. In the first step, each open call in P is replaced by its underapproximation (line 6) to obtain a closed program P', which is checked using a theorem prover (line 7). The routine *check* is a single call to the theorem prover. If a satisfying path is found, the algorithm terminates with BUG.

[2] In principle, any approximations may be used, but our implementation uses particular ones that are easy to compute.

The second step involves overapproximation. Let us ignore recursion bounding for the moment (assuming that the input program is non-recursive). Then $C_1 = C$ and $C_2 = \emptyset$. Line 11 replaces each open call by the overapproximation of its callee to obtain another closed program. If this program is correct, the algorithm terminates with CORRECT. Otherwise, all open calls on the counterexample τ are inlined and the algorithm repeats.

The procedure described so far (without recursion bounding) is not guaranteed to terminate, even for buggy programs. This is where recursion bounding comes to the rescue. For each call c, we define its *cost* to be the number of recursive calls necessary to reach c. (Note that calls in our setting refer to dynamic instances of the call, thus their cost is uniquely defined by their call stack.) When the cost of a call c reaches the recursion bound r, then c is always replaced by its underapproximation (in line 11). The function *split-on-cost*(C, r) partitions C into two sets C_1, C_2, where the latter has all those calls whose cost is r or greater. Once the search terminates with a particular bound, the bound is incremented to continue the search from where it stopped.

The main advantages of the stratified inlining algorithm are:

1. The program provided to the automated theorem prover is generated incrementally. Eager static inlining creates an exponential explosion in the size of the program. Stratified inlining delays this exponential explosion.
2. Stratified inlining inlines only those procedures that are relevant towards ruling out spurious counterexamples. Thus, it can often perform the search while inlining few procedures. This ability makes the search property-guided even outside the theorem prover. Because of the use of over-approximations, stratified inlining can be faster than static inlining even for correct programs.
3. If the program is buggy, then a bug will eventually be found, assuming that the theorem prover always terminates.

We now describe more formally how the verification conditions (VCs) are generated in Fig. 2. We first transform the procedures in the program to a collection of recursively-defined predicates; the predicate p will correspond to procedure `p`. The predicate p has four arguments—i, g, o', and g'. Intuitively, $p(i, g, o', g')$ means that `p` has an execution that begins in global state g and input i and terminates in global state g' and output o'. We create the definitions of these predicates in two steps. First, we introduce a fresh local variable $\_g$ to each procedure and replace each procedure call *call* $x := \mathtt{p}(e)$ with the sequence of statements: $\_g := g$; *havoc* g; *havoc* x; *assume* $p(e, \_g, x, g)$. Next, we convert the resulting call-free body of each procedure into a logical formula and define

$$Defn(p)(i, g, o', g') = \exists \bar{l}, \_g, o.\ \neg wp(Body(\mathtt{p}), \neg(g = g' \wedge o = o')).$$

where $wp(A, B)$ is the *weakest precondition* [3] of B with respect to A. Recall that our goal is to determine whether there is an execution of procedure `m` that begins in a state satisfying φ and ends in a state satisfying ψ. The algorithm maintains a formula P_f that corresponds to P in Fig. 2 and does the following to get the VCs:

- Line 1 sets P_f to the formula $\varphi(i,g) \wedge m(i,g,o',g') \wedge \psi(o',g')$. The set C is the set of all predicate occurrences in P_f.
- Before line 6, replace every occurrence of a predicate $p(t_1,t_2,t_3,t_4)$ in P_f with a fresh Boolean variable b. Let B be the set of all Boolean variables thus introduced and $Term(b)$ be the predicate occurrence corresponding to variable $b \in B$.
- Line 6 constructs the formula $P_f \wedge \bigwedge_{b\in B}(b \Leftrightarrow Term(b)) \wedge \bigwedge_{b\in B} \neg b$ and gives it to the theorem prover in line 7.
- If B_1, B_2 is the split of B that corresponds to the split C_1, C_2 of C, then line 11 constructs the formula $P_f \wedge \bigwedge_{b\in B}(b \Leftrightarrow Term(b)) \wedge \bigwedge_{b\in B_2} \neg b$.
- Let B' be the subset of all variables in B assigned *true* by the counterexample τ at line 15. Then at line 16, for each $b \in B'$, if $Term(b) = p(t_1,t_2,t_3,t_4)$, then substitute $Defn(p)(t_1,t_2,t_3,t_4)$ in place of b in P_f.

For ease of presentation, our algorithm was described with the restriction that there is a single global variable. The generalization to multiple global variables is straightforward and can be understood by simply thinking of g as a vector of variables in the entire preceding presentation. More interestingly, we also perform an important optimization based on the analysis of the modified variables by each procedure. We use a simple analysis to automatically annotate each procedure with the set of global variables modified by it. Then, *summary*(`p`) only needs to havoc the global variables modified by procedure `p`. The signature of the predicate $p(i,\bar{g},o,\bar{g}')$ becomes slightly more complicated though; while the vector $\bar{g}$ contains all global variables, the vector $\bar{g}'$ contains only the modified global variables.

3 Variable Abstraction and Refinement

Most often, many program variables are not needed for proving or disproving a given property. In this case, abstracting such variables can result in a huge improvement of the running time. However, abstraction can lead to an over-approximation of the behaviors of the original program and lead to spurious counterexamples. In this case, we use a refinement procedure to bring back some of the variables that were removed. We only apply variable abstraction to global variables. At any point in time, the set of global variables that are retained by the abstraction are called *tracked* variables.

Variable abstraction is also known as *localization reduction* [10] and has been used extensively in hardware verification. The variable abstraction that we describe here (Fig. 3) is the same as what was used in [12]. We also present the refinement algorithm (Alg. 1) used in [12]. Then we give two new refinement algorithms (Alg. 2 and Alg. 3) that perform significantly better than this one.

3.1 Variable Abstraction

Let *IsGlobalVar* be a predicate that is *true* only for global variables. Let *GlobalVars* be a function that maps an expression to the set of global variables that

$v := e$ $\mapsto \begin{cases} \text{assume true} & \mathit{IsGlobalVar}(v) \wedge v \notin T \\ \text{havoc } v & \mathit{GlobalVars}(e) \not\subseteq T \\ v := e & \text{otherwise} \end{cases}$	assume e $\mapsto \begin{cases} \text{assume true} & \mathit{GlobalVars}(e) \not\subseteq T \\ \text{assume } e & \text{otherwise} \end{cases}$
assert e $\mapsto \begin{cases} \text{assert false} & \mathit{GlobalVars}(e) \not\subseteq T \\ \text{assert } e & \text{otherwise} \end{cases}$	havoc v $\mapsto \begin{cases} \text{assume true} & v \notin T \\ \text{havoc } v & \text{otherwise} \end{cases}$

Fig. 3. Program transformation for variable abstraction, where T is the set of tracked variables.

appear in that expression. Let T be the current set of tracked variables. Variable abstraction is carried out as a program transformation, applied statement-by-statement, as shown in Fig. 3. Essentially, assignments to variables that are not tracked are completely removed (replaced by "assume true"). Further, expressions that have an untracked global variable are assumed to evaluate to any value. Consequently, assignments whose right-hand side have such an expression are converted to havoc statements.

3.2 Abstraction Refinement

Input: A set of tracked variables T
Input: A correct path P
Output: The new set of variables to track
1: **let** G be the set of global variables of P
2: $B := G - T$
3: $R := G$
4: *//Check which additional variables we need to track*
5: **for all** $v \in B$ **do**
6: $R := R - \{v\}$
7: $P' := \mathit{Abstract}(P, R)$
8: **if** $\mathit{check}(P')$ **then**
9: *//No need to track v*
10: skip
11: **else**
12: *//Abstracted too much, need to track v*
13: $T := T \cup \{v\}$
14: *//Check if we already have enough*
15: $P'' := \mathit{Abstract}(P, T)$
16: **if** $\mathit{check}(P'')$ **then**
17: **break**
18: **return** T

Fig. 4. Algorithm 1: The greedy refinement algorithm.

In Corral, we abstract the program using variable abstraction and then feed it to the stratified inlining routine. If it does not find a bug, then the input program is correct. If it returns a path, say τ, which exhibits an assertion failure in the abstract program, then we check to see if τ is feasible in the original program by *concretizing* it, i.e., we find the corresponding path in the input program. If this path is infeasible, then τ is a spurious counterexample. To rule out this counterexample, we must track some more variables. We now present three algorithms for carrying out refinement (each performs better than the previous one). The algorithms take a spurious counterexample as input and return a minimal set of variables to track that rule out the counterexample. The algorithms only look at the counterexample, and not the original program, nor the abstracted program.

We make use of two subroutines: *Abstract* takes a path and a set of tracked variables and carries out variable abstraction on the path. The subroutine *check* takes a path (with an assertion) and returns *true* only when the assertion can

be violated. We implement this by simply generating the VC of the entire path and feeding it to the theorem prover. Because paths have no recursion, we do not need to use the stratified inlining algorithm here.

The first algorithm, shown as Alg. 1, is the one used in [12]. It is a greedy algorithm. It takes as input the current set of tracked variables T and the counterexample P, represented as a (straight-line) program. The precondition is that P is correct (i.e., the counterexample is spurious). The output is the new set of variables to track. Alg. 1 works by iterating over all variables that were not tracked (line 5). For each such variable v, it checks if tracking v is really required (line 8). If so, it adds it to T (line 13) and then checks if it has enough tracked variables already (line 16). If so, it stops, otherwise, it repeats the loop.

Alg. 1 requires about $|G-T|$ number of iterations, and each iteration requires at least one call to *check*. In our experience, we have found that most spurious counterexamples can be ruled out by tracking one or two additional variables. The next two algorithms are contributions of this paper: both optimize for the case when only a few additional variables need to be tracked.

Alg. 2 uses a divide and conquer strategy. It has best case running time when only a few additional variables needs to be tracked, in which case the algorithm requires $\log(|G-T|)$ number of calls to *check*. In its worst case, which happens when all variables need to be tracked, it requires at most $2|G-T|$ number of calls to *check*. As our experiments show, most refinement queries tend to be towards the best case, which is an exponential improvement over Alg. 1.

Alg. 2 uses a recursive procedure *hrefine*. This procedure takes three inputs: T, D, and P, as described in the figure. It assumes that P, when abstracted with $G-D$, is correct. We refer to Alg. 2 as the top-level call *hrefine*$(T, \emptyset, P)$. Note that while *hrefine* is running, the arguments T and D can change across recursive calls, but P remains fixed to be the input counterexample.

Alg. 2 works as follows. If $T \cup D = G$ (line 1) then we already know that the program is correct while tracking T (because the precondition is that the program is correct while tracking $G-D$). Lines 3 to 5 check if T is already sufficient. Otherwise, in line 6, we check if only one variable remains undecided, i.e., $|G-(T \cup D)| = 1$, in which case the minimal solution is to include that variable in T (which is the same as returning $G-D$). Lines 8 to 11 form the interesting part of the algorithm. Line 8 splits the set of undecided variables into two equal parts randomly. Because of the check on line 6, we know that each of T_1 and T_2 is non-empty. Next, the idea is to use two separate queries to find the set of variables in T_1 (respectively, T_2) that should be tracked. The first query is made on line 9, which tracks all variables in T_2. The only remaining undecided variables for this query is the set T_1. When this returns, we know that all variables in S_1' should be tracked and all variables in $T_1 - S_1'$ should not be tracked. Thus, the second query includes S_1' in the set of tracked variables and $(T_1 - S_1')$ in the set of do-not-track variables. The procedure *hrefine* is guaranteed to return a minimal set of variables to track.

Theorem 1. *Given a program P with global variables G, and sets $T, D \subseteq G$, such that T and D are disjoint, suppose that Abstract$(P, G-D)$ is a correct*

program. If $R = hrefine(T, D, P)$ *then* $T \subseteq R \subseteq G - D$*, and* $Abstract(P, R)$ *is correct, while for each set* R' *such that* $T \subseteq R' \subset R$*,* $Abstract(P, R')$ *is buggy.*

The following Lemma describes the running time of the algorithm. It shows that if only a few additional variables need to be tracked, then the algorithm only requires a logarithmic number of calls to *check*.

Lemma 1. *If the output of Alg. 2 is a set* R*, then the number of calls to check made by the algorithm is* $O(|R - T| \log(|G - T|))$.

The following Lemma tightens the worst-case running time of the algorithm. This shows that Alg. 2 can be at most twice as slow as Alg. 1. Our experiments show that this worst-case is hardly ever reached in practice.

Lemma 2. *The number of calls to check made by Alg. 2 is bounded above by* $max(2|G - T| - 1, 0)$.

Procedure $hrefine(T, D, P)$
Input: A correct path P with global variables G
Input: A set of variables $T \subseteq G$ that must be tracked
Input: A set of variables $D \subseteq G$ that definitely need not be tracked, $D \cap T = \emptyset$
Output: The new set of variables to track

1: **if** $T \cup D = G$ **then**
2: **return** T
3: $P' := Abstract(P, T)$
4: **if** $check(P')$ **then**
5: **return** T
6: **if** $|G - (T \cup D)| = 1$ **then**
7: **return** $G - D$
8: $T_1, T_2 := partition(G - (T \cup D))$
9: $S_1 := hrefine(T \cup T_2, D, P)$
10: $S_1' := S_1 \cap T_1$
11: **return** $hrefine(T \cup S_1', D \cup (T_1 - S_1'), P)$

(*a*) **Algorithm 2**: $hrefine(T, \emptyset, P)$

Procedure $vcrefine(T, D, P)$
Input: A program P with special Boolean constants B
Input: A set of Boolean constants $T \subseteq B$ that must be set to *true*
Input: A set of Boolean constants $D \subseteq B$ that must be set to *false*
Output: The set of Boolean constants to set to *true*

1: **if** $T \cup D = B$ **then**
2: **return** T
3: $\psi = (\bigwedge_{b \in T} b) \wedge (\bigwedge_{b \in B - T} \neg b)$
4: **if** $check(\mathrm{VC}(P) \wedge \psi)$ **then**
5: **return** T
6: **if** $|B - (T \cup D)| = 1$ **then**
7: **return** $B - D$
8: $T_1, T_2 := partition(B - (T \cup D))$
9: $S_1 := vcrefine(T \cup T_2, D, P)$
10: $S_1' := S_1 \cap T_1$
11: **return** $vcrefine(T \cup S_1', D \cup (T_1 - S_1'), P)$

(*b*) **Algorithm 3**: $vcrefine(T, \emptyset, P)$

Fig. 5. Refinement algorithms based on divide-and-conquer.

While the *check* routine involves a (potentially expensive) theorem prover call, both the previous refinement algorithms spend a significant amount of time outside the theorem prover. The reason is that each iteration of the algorithm needs to abstract the path with a different set of variables, and then generate the VC for that path. In order to remove this overhead, the next algorithm that we present will require VC generation only once and the refinement loop will be carried out inside the theorem prover.

The first challenge is that a theorem prover does not understand our variable abstraction, thus, we cannot carry out the abstraction inside the theorem prover. We solve this by doing a *parameterized* variable abstraction as follows.

For each global variable v, we introduce a Boolean constant b_v. Next, we carry out the program transformation shown in Fig. 6. The transformed program has the invariant: if b_v is set to *true* then the program behaves as if v is tracked, otherwise it behaves as if v is not tracked. The transformation uses a subroutine *Tracked*, which takes an expression e as input and returns a Boolean formula:

$$\mathit{Tracked}(e) = \bigwedge_{v \in \mathit{GlobalVars}(e)} b_v$$

Thus, *Tracked*(e) returns the condition under which e is tracked.

If P was the input counterexample and T the current set of tracked variables, then we transform P to, say, P' using parameterized variable abstraction. Next, we set each b_v, $v \in T$ to *true*. Let B be the set of Boolean constants b_v, $v \notin T$. The refinement question now reduces to: what is the minimum number of Boolean constants in B that must be set to *true* so that P' is correct, given that setting all constants in B to *true* makes P' correct? We solve this using Alg. 3, which takes P' as input.

Alg. 3 is exactly the same as Alg. 2 with the difference that instead of operating at the level of programs and program variables, it operates at the level of formulae and Boolean constants. This buys us a further advantage: the queries made to the theorem prover on line 6 are very similar. One can save VC(P) on the theorem prover stack and only supply ψ for the different queries. This enables the theorem prover to reuse all work done on VC(P) across queries.

```
if(¬Tracked(v)) {
  assume true;
} else if(¬Tracked(e)) {
  havoc v;
} else {
  v := e;
}
```

(a)

```
if(¬Tracked(e)) {
  assume true;
} else {
  assume e;
}
```

(b)

```
if(¬Tracked(e)) {
  assert false;
} else {
  assert e;
}
```

(c)

Fig. 6. Program transformation for parameterized variable abstraction: (*a*) Transformation for $v := e$; (*b*) assume e; (*c*) assert e. Other statements are left unchanged.

4 Experiments

We ran Corral on a collection of Windows device drivers. For each driver, we had various different harnesses that tested different functionalities of the driver. Some of the harnesses were concurrent, in which case the sequential program was obtained using a concurrent-to-sequential source transformation described elsewhere [6, 14]. These drivers also had planted bugs. The drivers are written in C. We use HAVOC [8] to compile them to Boogie.

A summary of these drivers and Corral's running time is shown in Fig. 7. We report: the number of non-empty, non-scoping C source lines (LOC), the number of global variables in the generated Boogie file (Vars), the number of procedures (Procs), whether the driver is concurrent (Conc?), whether it has a

Name	LOC	Vars	Procs	Conc?	Correct?	Iter	Time (sec)		
							Total	R(%)	S(%)
daytona	660	114	40	Yes	Yes	8	26.9	50	35
daytona_bug2	660	114	40	Yes	No	6	27.0	56	27
kbdclass_read	978	212	48	Yes	Yes	12	194.4	52	29
mouclass_read	818	179	44	Yes	Yes	13	185.7	53	28
mouclass_bug3	818	179	44	Yes	No	15	245.5	53	30
pcidrv_ioctl	661	109	49	Yes	Yes	12	54.5	43	46
serial_read	1601	378	77	Yes	Yes	13	1151.7	41	51
mouser_sdv_a	3311	225	131	No	No	4	35.4	33	45
mouser_sdv_b	3898	252	143	No	Yes	12	990.8	15	81
fdc_sdv	5799	421	180	No	Yes	11	659.8	11	85
serial_sdv_a	7373	466	149	No	Yes	6	139.2	47	34
serial_sdv_b	7396	439	168	No	No	6	289.1	46	40

Fig. 7. Running times of CORRAL on driver benchmarks.

planted bug or not (Correct), the number of iterations of the refinement loop (Iter), the total running time of CORRAL (Total), the fraction of time spent in abstraction and refinement (R%) and the fraction of time spent in checking using the stratified inlining algorithm (S%). The refinement algorithm used was Alg. 3. CORRAL fares reasonably well on these programs. A significant fraction of the time is spent refining, thus, justifying our investment into faster refinement algorithms. CORRAL is about 4 times slower on average when we use Alg. 2. A more detailed comparison of refinement algorithms follows later in this section.

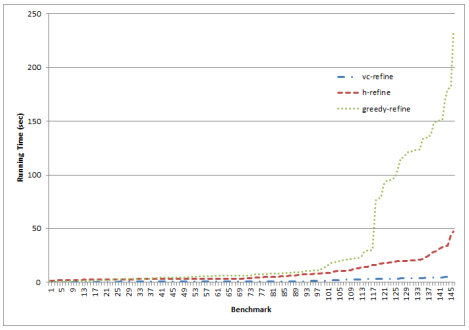
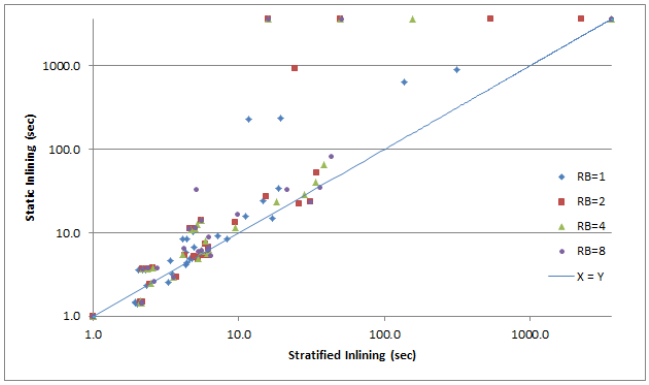


Fig. 8. A comparison of running times of various refinement algorithms.

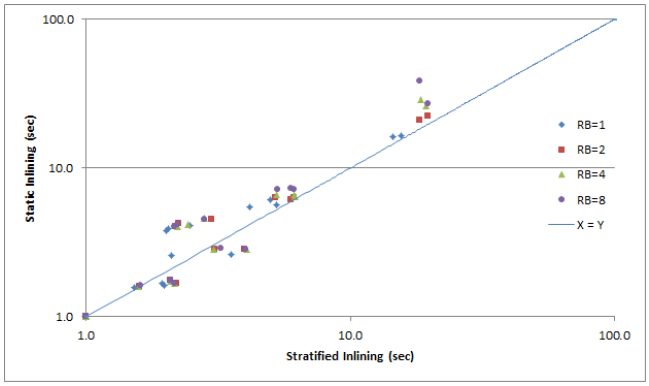
Fig. 9 shows a comparison of the running time of stratified inlining versus static inlining. Each dot in the figures corresponds to a query on different program. The experiments are classified according to the outcome (correct or buggy) and the maximum recursion bound (RB). Static inlining ran out of memory on each of the six largest programs, with recursion bound of 1 or 2. The timing results shown are for ones in which static inlining did not run out of memory.

The buggy instances were small, and the difference between the two techniques is not much. For the correct instances, the comparison is unclear for small instances (that took less than 10 sec), but as the running times become larger, stratified inlining significantly outperforms static inlining, with the latter timing out in many cases (the timeout is set to 3600 sec).

A third experiment, shown in Fig. 8, compares the different refinement schemes. Each dot in the figure is a different counterexample. All three algo-



(a) Correct programs



(b) Buggy programs

Fig. 9. A comparison of running times of static inlining versus stratified inlining for different values of the recursion bound (RB)

rithms agreed on the outcome in each case. It is clear from the figure that Alg. 3 is far superior to the other two algorithms. Moreover, its performance is quite robust because its running time does not vary as drastically as for the other ones. The average speedup of Alg. 3 over Alg. 2 was 2.8X and that over Alg. 1 was 13.2X.

5 Related Work

The idea behind stratified inlining is similar to previous work on *structural abstraction* [1] and *inertial refinement* [17]. However, the former does not use an underapproximation by blocking certain calls. The latter uses both over and under approximations to iteratively build a view of the program that is then an-

alyzed. A distinguishing factor is our use of recursion bounding as well as using inlining to construct a single VC for the entire program view.

The work of Liang et al. [15] uses a divide-and-conquer scheme to learn minimal abstractions, similar to our refinement algorithms. The problems are similar: in each case, one wants to establish the smallest reason that can prove a certain goal. However, Liang et al. operate in a machine-learning setting and use a random walk over the set of parameters to find the minimal set. Our setting is verification, and our algorithms guarantee to find the minimal set. Moreover, the use of verification conditions enabled the design of Alg. 3.

References

1. D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification*, 2007.
2. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, 2001.
3. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis for Software Tools and Engineering*, 2005.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.
5. Boogie: An Intermediate Verification Language. `http://research.microsoft.com/en-us/projects/boogie/`.
6. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Principles of Programming Languages*, 2011.
7. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Foundations of Software Engineering*, 2006.
8. HAVOC. `http://research.microsoft.com/en-us/projects/havoc/`.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, 2002.
10. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata Theoretic Approach*. Princeton University Press, 1995.
11. S. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages*, 2008.
12. S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification*, 2009.
13. A. Lal, S. Qadeer, and S. Lahiri. Corral: A whole-program analyzer for boogie. Technical Report MSR-TR-2011-60, Microsoft Research, 2011.
14. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1), 2009.
15. P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *Principles of Programming Languages*, 2011.
16. Poirot: The Concurrency Sleuth. `http://research.microsoft.com/en-us/projects/poirot/`.
17. N. Sinha. Modular bug detection with inertial refinement. In *Formal Methods in Computer Aided Design*, 2010.
18. Z3 Theorem Prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.