Preservation of proof obligations for hybrid verification methods

Gilles Barthe, César Kunz, David Pichardie, Julian Samborski-Forlese

To cite this version:


HAL Id: inria-00332718
https://hal.inria.fr/inria-00332718
Submitted on 21 Oct 2008
Preservation of proof obligations for hybrid verification methods

Gilles Barthe
IMDEA Software, Spain
Gilles.Barthe@imdea.org

César Kunz
INRIA Sophia Antipolis, France
Cesar.Kunz@inria.fr

David Pichardie
INRIA Rennes/IRISA, France
David.Pichardie@irisa.fr

Julián Samborski-Forlese
IMDEA Software, Spain
Julian.SF@imdea.org

Abstract

Program verification environments increasingly rely on hybrid methods that combine static analyses and verification condition generation. While such verification environments operate on source programs, it is often preferable to achieve guarantees about executable code. We show that, for a hybrid verification method based on numerical static analysis and verification condition generation, compilation preserves proof obligations and therefore it is possible to transfer evidence from source to compiled programs. Our result relies on the preservation of the solutions of analysis by compilation; this is achieved by relying on a bytecode analysis that performs symbolic execution of stack expressions in order to overcome the loss of precision incurred by performing static analyses on compiled (rather than source) code. Finally, we show that hybrid verification methods are sound by proving that every program provable by hybrid methods is also provable (at a higher cost) by standard methods.

1 Introduction

Program verification techniques are widely used to reason about the correctness of applications and play an important role in fields such as mobile code and embedded systems where strong guarantees are required. However, program verification, and in particular deductive program verification, is traditionally applied to source code, whereas guarantees are often required about executable code. This discrepancy is particularly acute in the context of mobile code, where code consumers do not trust code producers and may not have access to the source code. Therefore, it is of interest to develop methods to transfer evidence from source code verification to the code consumers. In earlier work [3], we focussed on transferring proofs from source to compiled programs in the context of verification methods based on verification condition generators, that are commonly used in Proof Carrying Code [15] and program verification environments. We have shown that non-optimizing compilation preserves proof obligations, and therefore that the certificates of source programs can be reused directly to validate compiled programs. However, state-of-the-art verification tools do not use plain verification condition generation; instead, they rely on hybrid methods, that combine static analyses and verification condition generation.

The objective of the paper is to extend preservation of proof obligations to hybrid verification methods. For concreteness, we consider a small imperative language with arrays, and we focus on a hybrid method based on a generic numerical analysis [13, 5] and that can be instantiated to several numeric domains, including polyhedra.

We first define a hybrid verification method in which programs are subjected to static analysis, and then to verification condition generation. The VCgen exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded: more precisely, the VCgen ignores the case of out-of-bound accesses whenever the analysis ensures that accesses are within bounds. This leads to fewer, smaller verification conditions. Furthermore, the VCgen adds the results of the analysis as additional assumptions to help the user prove the verification conditions. This is particularly useful for the relational analyses considered as they can provide part of the invariants required to prove programs correct.

Then, we prove preservation of proof obligations using the techniques of Barthe et al. [3]. The proof relies on knowing that the solutions of the analysis are preserved by compilation. Although analyzing compiled programs is known to be less precise than analyzing source programs, as
stated by Logozzo and Fähndrich [12], we achieve preservation of solutions by defining at bytecode level an analysis that performs a symbolic execution of stacks [21, 20, 5].

Finally, we relate hybrid verification to standard verification. We show that programs that are provably correct using our hybrid method, remain provably correct using standard verification condition generation; to this end, we define a compiler that transforms a hybrid specification (combining logical assertions and analysis results) into a logical one by giving a logical interpretation of the analysis results.

2 Proof Carrying Code and Hybrid Methods

The general goal of our work [2] is to develop efficient, scalable and trustworthy security architectures for mobile code. We adopt the view of Proof Carrying Code (PCC) [15], which emphasizes security through verifiable evidence, and require that mobile code components come equipped with a certificate that can be checked efficiently and independently by the code consumer to ensure that the downloaded components issued by the producer respect its policy.

There are two main flavours of PCC, that respectively rely on type systems and program verification. We briefly outline the issues of certificate generation and certificate checking for both of them.

In the type-based approach, certificates are generated automatically by an analyzer and packaged with the code; on the consumer side, the certificate is verified efficiently by a checker that is tightly coupled with the analyzer (in the spirit of lightweight bytecode verification). In the logic-based approach, certificates may be generated automatically by certifying compilers, provided the policies are sufficiently simple. However, certificates must be constructed interactively once policies become more complex; preservation of proof obligations allows the construction of the certificate to arise at source level. The certificate (that may include some program annotations) is packaged with the program. On the consumer side, the protocol proceeds in three steps: first, annotations are checked against the policy; second, verification condition generation is used to generate proof obligations; third, certificate checking is performed using a proof checker, that verifies whether the certificates prove the proof obligations.

Hybrid methods aim to provide a tight integration of type systems and logical methods. There are two approaches to hybrid program verification. In the explicit approach, the user provides safety annotations that are used by the verification condition generator, and checked by an annotation checker. In contrast, the implicit approach advocates that the annotations are inferred by a static analyzer, and then used by verification condition generation. Both approaches are used (sometimes in conjunction) in deductive program verification, as well as in type-based analyses.

For the hybrid method developed in this paper, program verification proceeds as follows: first, the annotated program is subjected to the static analysis, and a solution is computed (note the analysis does not take advantage of the annotations). Then, the verification condition generator uses the solution to compute a smaller set of proof obligations, which must then be discharged interactively. On the consumer side, the program arrives packaged with the solution of the analysis (or as in lightweight bytecode verification a partial solution that contains sufficient information to recompute the solution efficiently), the program annotations, and the certificate, and the checking proceeds in four steps; first, one checks the correctness of the analysis. The remaining three steps are as in logic-based PCC.

3 Setting

This section introduces the source language (an imperative language with arrays of integers), the target language (a stack-based language with jumps), and the compiler.

We assume given two disjoint sets \( V_s \) of scalar variables and \( V_a \) of array variables, and let \( V \) denote \( V_s + V_a \). Each variable in \( V_a \) has an associated size. Furthermore, we assume given two sets \( V_{s,old} \) and \( V_{a,old} \) in 1-1 correspondence with \( V_s \) and \( V_a \), which are used to store initial values. We also consider a special variable \( \text{res} \), which is used to represent the value of the program result. Finally, we assume given a set \( \text{Lab} \subset \mathbb{N} \) of labels.

Source Language

Programs are defined as commands, and are decorated with labels in order to express analysis results:

\[
\begin{align*}
e & ::= e \ op \ e \ | \ n \ | \ x \ | \ a[e] \\
c & ::= \text{Skip} \ | [x := e]_k \ | [a[e] := e]_k \ | c ; c \ | \text{return} e)_k \\
& \text{if } [e \times e]_k \text{ then } c \text{ else } c \ | \text{while } [e \times e]_k \text{ do } c
\end{align*}
\]

where \( x, a, n \) and \( k \) respectively range over \( V_s, V_a, \mathbb{Z} \) and \( \text{Lab} \), \( \text{op} \) ranges over (binary) arithmetic operations, and \( \times \) over arithmetic comparisons. We assume that labels occur at most once in commands.

The semantics of source programs is formalized by a small-step transition relation between states. States may be intermediate, in which case they consist of a statement and of a memory, or final, in which case they consist of a memory, and possibly a tag to denote abnormal termination. Memories are modeled as pairs of mappings respectively from variables to values and from arrays to indices to values. We assume that each array \( a \) comes equipped with its size \( |a| \) and define the semantic domains of the source
language as follows:

\[
\begin{align*}
V_{Mem} &= V_a \rightarrow \mathbb{Z} \\
A_{Mem} &= \Pi a \in V_a. \{i | 1 \leq i \leq |a|\} \rightarrow \mathbb{Z} \\
Mem &= V_{Mem} \times A_{Mem} \\
State^i_F &= Stmt \times V_{Mem} \times A_{Mem} \\
State^a_F &= V_{Mem} \times A_{Mem} \times (\mathbb{Z} + \{\text{AOB}\}) \\
State^a &= State^i_F + State^a_F
\end{align*}
\]

The operational semantics of programs is standard and, thus, omitted. (See the next subsection for the semantics of instructions that manipulate arrays.)

**Bytecode Language**

A bytecode program is defined as a list of instructions. Instructions either manipulate the memory that stores the values of variables and the contents of arrays, or manipulate the operand stack, or perform a conditional or unconditional jump. The set of instructions is defined by the grammar

\[
\text{ins ::= prim op | push v | load x | store x | return | aload a | astore a | cjmp \ x l | jmp l | nop}
\]

We denote by \( p[l] \) the instruction at position \( l \) of a bytecode program \( p \). The semantics of bytecode programs is formalized using a transition relation between states. States may either be intermediate or final; intermediate states consist of a program counter, an operand stack, that stores the results of intermediate computations, and a memory. The semantic domains of the bytecode language are defined as follows, where we implicitly assume that the program counter is within the bounds of programs:

\[
\begin{align*}
Stack &= \mathbb{Z}^* \\
State^i_F &= \mathbb{N} \times V_{Mem} \times A_{Mem} \times Stack \\
State^a_F &= V_{Mem} \times A_{Mem} \times (\mathbb{Z} + \{\text{AOB}\}) \\
State^a &= State^i_F + State^a_F
\end{align*}
\]

The operational semantics of programs is standard. We only provide the operational semantics of the instructions aload and astore; these instructions may cause abrupt termination if array accesses are out-of-bound. The rules are given in Figure 2, where we use the notation \( [f | s \rightarrow r] \) to refer to the function that is identical to \( f \) everywhere except in \( r \) that returns \( s \), for any sets \( R \) and \( S \) and any function \( f : R \rightarrow S \).

**Compiler**

The compiler is standard, and defined in Figure 1; we use the function \( \text{init} : \text{Stm} \rightarrow \text{Lab} \) to associate to each statement its initial label. We assume label compatibility, i.e., that the label of a source statement is the same as the label of the program point for its compilation.

Throughout the rest of the paper we let \( P \) be a source program, and the bytecode program \( \hat{p} \) the result of the compilation of program \( P \).

**Figure 1. Compiler**

\[
\begin{align*}
P[i] &= \text{aload a} \quad 0 \leq n < |a| \\
\langle i, \rho_v, \rho_a, n : s \rangle &\rightarrow \langle i+1, \rho_v, \rho_a, a n : s \rangle \\
P[i] &= \text{aload a} \quad -0 \leq n < |a| \\
\langle i, \rho_v, \rho_a, n : s \rangle &\rightarrow \langle \rho_X, \rho_a, \text{AOB} \rangle \\
P[i] &= \text{astore a} \quad 0 \leq n < |a| \\
\langle i, \rho_v, \rho_a, n : v : s \rangle &\rightarrow \langle i+1, \rho_v, \rho_a, a n : a n : v, s \rangle \\
P[i] &= \text{astore a} \quad -0 \leq n < |a| \\
\langle i, \rho_v, \rho_a, n : v : s \rangle &\rightarrow \langle \rho_X, \rho_a, \text{AOB} \rangle \\
P[i] &= \text{return} \\
\langle i, \rho_v, \rho_a, n : s \rangle &\rightarrow \langle \rho_v, \rho_a, n \rangle
\end{align*}
\]

**Figure 2. Semantics of bytecode (excerpts)**

4 Preservation of solutions

It is folklore that compilation potentially yields a loss of precision for relational analyses. The purpose of this section is to show that solutions of abstract interpretations are preserved by compilation, provided one uses symbolic expressions, as done in [21, 20, 5], to mitigate the presence of the operand stack and to recover the loss of precision incurred by compilation.

**Symbolic Expressions**

Expressions and guards serve as the interface with the numerical relational domain in the analysis for bytecode. Be-
low we let \( x \) range over \( V \).

\[
Expr \ni e ::= \ n \mid x \mid x[e] \mid ?[e] \mid e \ op \ e \quad x \in V
\]

The expression \( ? \) represents an unknown value; therefore, expressions are interpreted as sets of possible values. Formally, the semantics \([e]_\rho\) and \([t]_{\rho_1}^{\rho_2}\) of expressions with respect to an environment \( \rho = \langle \rho_v, \rho_a \rangle \) are defined by the clauses:

\[
\begin{align*}
[n]_\rho &= \{n\} \\
[x]_\rho &= \rho_v x \\
[?]_\rho &= Z \\
[?e]_\rho &= Z \\
[x[e]]_\rho &= \{\rho_a x v \mid v \in [e]_\rho\} \\
[e_1 op e_2]_\rho &= \{n_1 op n_2 \mid n_1 \in [e_1]_\rho, n_2 \in [e_2]_\rho\} \\
[e_1 \times e_2]_\rho &= \exists n_1 \in [e_1]_\rho, n_2 \in [e_2]_\rho s.t. n_1 \times n_2
\end{align*}
\]

Note that the expression \( ? \) is not required for analyzing bytecode programs that are achieved by compilation of the source program, since the stack is empty after storing a value in an array. However, it provides more precision when dealing with programs that are not obtained by compilation.

### Abstract Domain

Following Miné [13], we assume given an abstract numerical domain interface, which can be instantiated with standard relational abstract domains. The interface consists of a domain \( D \) equipped with a partial order \( \subseteq \subseteq D \times D \), meet and join operators \( \cap, \cup : D \times D \rightarrow D \), a least element \( \bot \) and a greater element \( \top \). We also assume given abstract assignment functions \([x:=e]_\rho^D, [x[e]:=e_2]_\rho^D : D \rightarrow D\text{, and a function } assume^D \text{ that maps guards to abstract elements.}

Finally, we assume given a monotone concretization function \( \gamma : D \rightarrow P(V\text{Mem }\times\text{AMem}) \) mapping abstract elements to sets of environments in \( V\text{Mem }\times\text{AMem} \), and satisfying the properties in Figure 3.

We define the abstract test \([t]_\rho^D : D \rightarrow D\text{ of a guard } t \in \text{Guard by } [t]_\rho^D(t') = assume^D(t) \cap t'\text{.}

### Source Code Analysis

The source code analysis is specified by abstract transfer functions that map elements of the abstract domain into elements of the abstract domain.

**Definition 1 (Abstract Domain for High-Level)** A result of the analysis for the source program \( P \) is described by a mapping \( \text{Loc} \) in the lattice \( \text{State}^D = \text{Lab} \rightarrow D\).  

**Definition 2 (Solution)** A mapping \( \text{Loc} \) for the source program \( P \) is a solution of the analysis if it verifies the constraint system defined in Figure 4, i.e. \( \text{Loc} \vdash P \) holds.

### Byte Code Analysis

As for the source code analysis, the bytecode analysis is defined by abstract transfer functions that map abstract states into abstract states. In this case, the abstract states are pairs of the form \( (s^b, l^b) \) where \( l^b \) is an element of the abstract domain, and the list of symbolic expressions \( s^b \) abstracts the operand stack. The symbolic abstract domain for stacks is \( Expr^* \), where for any set \( A, A^* \) denotes the domain of lists with elements in \( A \). The set of variables considered by the bytecode analysis is the same as in the source code analysis.

**Definition 3 (Bytecode Abstract Domain)** A result of the analysis for \( \hat{p} \) is described by a mapping \( \text{loc} \) in the lattice \( \text{state}^D = \text{Lab} \rightarrow (Expr^*_L \times D)\).  

An analysis result is a solution of the analysis if it satisfies the constraint system associated to each program. The constraint system is defined in Figure 5. For instructions other than branching or return instructions, the constraint is defined by partial transfer functions in \( Expr^* \times D \rightarrow (Expr^* \times D) \), most of them defined as a symbolic execution affecting the abstract representation of the operand stack.

**Definition 4 (Solution)** A mapping \( \text{loc} \) for the bytecode program \( \hat{p} \) is a solution of the analysis if it satisfies the constraint system of Figure 5, i.e. \( \text{loc} \vdash \hat{p} \) holds.

### Preservation of Solutions

We define first the compilation of a source code analysis solution and then show that it is a solution for the bytecode analysis. For notational convenience, we denote by \( f_{s_1, \ldots, s_n}(s^b, l^b) \) the composition \( f_{s_n} \cdots f_{s_1}(s^b, l^b) \cdots \), where \( s_1, \ldots, s_n \) is a sequence of bytecode instructions. Let \( \text{succ}(l) \) denotes the set of successors of a label \( l \), e.g. \( \text{succ}(l) = \emptyset \) and \( \text{succ}(l) = \{l + 1, l'\} \) respectively for \( \hat{p}[l] = \text{return} \) and \( \hat{p}[l] = \text{cmp} \times l' \). The set \( \text{pred}(l) \) is defined as \( \{l' \mid l \in \text{succ}(l')\} \).
For each byte code program $\hat{p}$, we can extract from the previous constraint system a set of transfer functions $(\hat{g}_{i,j})_{(i,j) \in \text{Lab}^2}$ such that $\hat{\text{loc}} \vdash \hat{p}$ if and only if $\bigcup_{k' \in \text{pred}(k)} \hat{g}_{i',k}(\hat{\text{loc}}(k')) \subseteq \hat{\text{loc}}(k)$ for all $k \in \text{dom}(\hat{p})$.

We can extend a partial function $\hat{\text{loc}}_{\text{partial}} \in \text{state}^5$ to a total function $\hat{\text{loc}}$ on $\text{dom}(\hat{p})$ if we set

$$\hat{\text{loc}}(k) = \begin{cases} \hat{\text{loc}}_{\text{partial}}(k) & \text{if } k \in \text{dom}(\hat{\text{loc}}_{\text{partial}}) \\ \bigcup_{k' \in \text{pred}(k)} \hat{g}_{i',k}(\hat{\text{loc}}(k')) & \text{otherwise} \end{cases}$$

This definition only makes sense if, by considering the control flow graph of $\hat{p}$ whose edges are $\{(i, j) \mid i \in \text{dom}(\hat{p}) \land j \in \text{succ}(i)\}$, every loop contains a label in $\text{dom}(\hat{\text{loc}}_{\text{partial}})$. We refer to the function $\hat{\text{loc}}$ as the completion of the partial function $\hat{\text{loc}}_{\text{partial}}$.

Definition 6 (Compiled analysis results) Given an analysis result $\text{Loc}$ for the program $P$, an analysis result compiled from $\text{Loc}$ is the completion of the function $\hat{\text{loc}}_{\text{partial}}$ defined on each $k \in \text{dom}(\text{Loc})$ by $\hat{\text{loc}}_{\text{partial}}(k) = ([], \text{Loc}(k))$.

This definition can be shown to be well defined from the facts that $\text{Loc}$ annotates every loop in $P$ and each loop in the control flow graph of $\hat{p}$ contains a label of a loop in $P$.

Lemma 7 Let $\hat{p}_1, \hat{p}_2$ and $e$ s.t. $\hat{p} = \hat{p}_1 :: l : [e]_e :: l' : \hat{p}_2$. Then, $\hat{\text{loc}}(l') = f_{\text{inst}}(s^2, l') = (e : s^2, l')$ where $(s^2, l') = \hat{\text{loc}}(l)$ and $[1; \ldots; i_k] = [e]_e$.

**PROOF.** We prove the lemma by structural induction over expression $e$.

The following lemma states the main result of this section: compilation preserves analysis solutions.

Lemma 8 If $\text{Loc}$ is s.t. $\text{Loc} \vdash P$, then the analysis result $\hat{\text{loc}}$ compiled from $\text{Loc}$ is s.t. $\hat{\text{loc}} \vdash \hat{p}$, i.e. it is a solution of
the bytecode analysis.

PROOF. By structural induction over s we can prove that
if \( p = p_1 :: k_1 : [s] :: k_2 : p_2 \) and exists \( t^\beta \) such that
\[
\text{Loc} \vdash \{ \text{Loc}(k_1) \} s \{ t^\beta \} \quad \text{and} \quad t^\beta \subseteq \text{l\text{o}c}(k_2)
\]
then we have: \( \forall k \in [k_1, k_2), \, \text{l\text{o}c} \vdash k : p[k] \).

5 Preservation of proof obligations

In this section we define two verification frameworks, respectively for source programs and for unstructured bytecode of previous sections. As a specification language we consider first order formulae, namely the domain of assertions \( A \). The validity of an assertions in a particular execution state \( \eta \in \text{State}^\text{e} \) is standard. In particular, an assertion that contains the expression \( a[e] \) is invalid in those execution states in which \( e \) is out of the bounds of the array \( a \).

We consider a program specification a tuple \((\text{pre}, \text{annot}, \text{post}, \chi)\), where the assertion \text{pre} is a precondition, post and post are respectively normal and abnormal postconditions, and the partial function \text{annot} : \text{Lab} \rightarrow A maps program labels to internal points specifications. The special variable \text{res} may only occur in post, and \text{pre} only refers to variables from \( V \). When specifying a bytecode program, assertions may refer to the special variable \( s \) representing the operand stack.

We say that a program satisfies the specification \((\text{pre}, \text{annot}, \text{post}, \chi)\) if every execution starting in a state that satisfies \text{pre} only reaches normal final states satisfying post or abnormal states satisfying \( \chi \), and only reaches intermediate \( l \)-labeled points satisfying \text{annot}(l). Given a program specification \((\text{pre}, \text{annot}, \text{post}, \chi)\), a verification condition generator \( \text{VCgen} \) framework provides a set of sufficient proof obligations that ensures that the program satisfies the specification.

The VCgens defined in this section are hybrid in the sense that they take advantage of a previously computed analysis to reduce the size of proof obligations. We assume that the result of a relational analysis \((\text{Loc} \text{ and } \text{l\text{o}c}\text{ respectively for source and bytecode programs})\) is given as input to the VCgen. For the abstract domain \( D \), we consider a relation \( \models \subseteq D \times A \) such that for any guard \( b \) and any \( d \in D, \, d \models b \) indicates that the interpretation of the abstract element \( d \) ensures the validity of the condition \( b \). For example, when accessing an array in the expression \( a[x] \) we shall check that the value of the variable \( x \) is within the bounds of the array \( a \). If we instantiate \( D \) with the domain of convex polyhedra, each element \( d \in D \) represents a set of linear constraints from which we can discover whether the condition \( 0 \leq x < |a| \) is satisfied.

A further improvement over standard VCgens consists of reusing the result of the analysis to strengthen loop invariants. This technique helps reducing the size of annotations and the burden of interactive specification. To that end, we assume a concretization function \( \gamma_a : D \rightarrow A \) to interpret abstract elements \( d \in D \) as assertions.

VCgen for Source Programs

Consider a specification \((\text{pre}, \text{annot}, \text{post}, \chi)\) for the source program \( P \). In this section, we assume that \( \text{annot} \) sufficiently annotates the program \( P \), that is, for every subprogram \( \text{while } [t] \text{ do } c \text{ of } P \), we have that \( l \in \text{dom}(\text{annot}) \).

A VCgen for source programs is defined by the set of proof obligations \( \text{PO} \) defined as \( \{ \text{pre} \Rightarrow \phi[V_{\text{old}}] \cup \theta \} \), where \( \phi, \theta = \text{WP}(P, \text{post}), \phi[V_{\text{old}}] \) represents the result of substituting in \( \phi \) any array or scalar variable \( x_{\text{old}} \) in \( V_{\text{old}} \) by \( x \), and the function \( \text{WP} \) is defined in Figure 6. In the figure, the assertion \( \text{inB}(e) \) stands for the condition that must satisfy an execution state to ensure that every array access in \( e \) is within bounds. For instance, if \( e \) does not contain array expressions \( \text{inB}(e) \) is defined as true and \( \text{inB}(a[e]) \) as \( 0 \leq e < |a| \). We follow the simplifying assumption that expressions contain no more than one array access. For any array variable \( a \) and expressions \( e_1 \) and \( e_2 \), \( \text{upd}(a, e_1, e_2) \) is interpreted as the array \( a' \) such that \( a'[e] \) is evaluated to \( e_2 \) if \( e_1 = e \) and to \( a[e] \) otherwise. To simplify the presentation of examples, proof obligations for \( \text{while} \) statements are split into two assertions corresponding to the true and false branches.

The function \( \text{WP} \) considers the result of the analysis \( \text{Loc} \) to reduce the size of proof obligations. That is, if the abstract value \( \text{Loc}(l) \) associated to the program point under consideraction indicates that any array access in the statement is within bounds, the returned predicate is simplified by omitting the exceptional postcondition. Consider the program of Figure 7. If the analysis is able to compute at label \( k_1 \) an abstract value \( d \) such that \( d \models 0 \leq i < |A| \), the \( \text{WP} \) function will return the assertion \( \text{upd}(A, i, A[0])[i + 1 - 1] = A[0] \), which together with the loop invariant at label \( k \) yields the proof obligation:

\[
A[i - 1] = 0 \land 0 \leq i \leq |A| \quad \Rightarrow \quad i < |A| \Rightarrow \text{upd}(A, i, A[0])[i + 1 - 1] = A[0]
\]

where the boxed assertion \( 0 \leq i < |A| \) represents the result of the analysis at the loop entry point.

In contrast, if we do not take advantage of the result of the analysis we must prove the bigger formula:
As can be seen from the definition of WP, proof obligations are of the form \( \phi_1 \land \gamma_\delta(d) \Rightarrow \phi_2 \), whereas a standard VCgen outputs the stronger proof obligation \( \phi_1 \Rightarrow \phi_2 \). In consequence, one can provide the code with a weaker invariant \( \phi_1 \) as long as the analyzer is able to eventually infer the missing information \( \gamma_\delta(d) \). For instance, for the simple program of Figure 7, a standard VCgen will return the invalid proof obligation

\[
A[i - 1] = A[0] \land 0 \leq i \leq |A| \Rightarrow \\
i < |A| \Rightarrow \\
(0 \leq i < |A| \Rightarrow \text{upd}(A, i, A[0])[i + 1 - 1] = A[0] \land \neg(0 \leq i < |A|) \Rightarrow \text{false})
\]

for the path that does not enter the loop. It is sufficient to provide a stronger invariant, i.e. in conjunction with the condition \( i \leq |A| \), to prove the program correct. However, as an alternative to increasing the size of the program annotations, assuming the condition \( i \leq |A| \) is inferred by the analysis, the hybrid VCgen generates the weaker (and valid) proof obligation:

\[
A[i - 1] = A[0] \land 0 \leq i \leq |A| \Rightarrow \\
\]

**Preservation of Proof Obligations**

Consider the specification \( \text{pre}, \text{annot}, \text{post}, \chi \) for source program \( P \), and assume that \( \text{annot} \) is a sufficient annotation for \( P \), i.e., every loop is annotated. Let \( \text{pre}, \text{annot}, \text{post}, \chi \) define as well the specification for the bytecode program \( \hat{p} \). From previous results, we know that if \( \text{annot} \) is a sufficient annotation for \( P \) then it is also a sufficient annotation for the result of the compilation \( \hat{p} \). Let \( \text{Loc} \) be a solution of the analysis for the source program \( P \), and \( \text{loc} \) a solution of the analysis for the bytecode program \( \hat{p} \), compiled from \( \text{Loc} \) as described in Section 4.

We assume that the concretization functions satisfy the property \( \gamma_\delta([[1]], d) = \gamma_\delta(d) \), so that the interpretation of abstract analysis results in the source and bytecode sides coincides (recall that by definition \( \text{loc}(l) = ([1], \text{Loc}(l)) \) for every \( l \) in \( \text{dom}(\text{Loc}) \)). In addition, for any expression \( e \) and any \( d \in D \), if \( e \) does not contain array expressions, i.e. \( \text{inB}(e) = \text{true} \), then \( d \models \text{inB}(e) \).

The following result about the compilation of expressions is helpful to prove preservation of proof obligations:

**Lemma 9** Let \( \hat{p} \) be equal to \( \hat{p}_1 :: l_1: [e], \hat{p}_2: l_2 : \hat{p} \). Then \( \text{wp}(l_1) \) is equal to \( \text{wp}(l_2)[e:\gamma_\delta] \) if \( \text{loc}(l_1) \models \text{inB}(e) \) and equal to \( \text{wp}(l_2)[e:\gamma_\delta] \land \neg \text{inB}(e) \Rightarrow \chi \) otherwise.

The coincidence of the sets of proof obligations \( PO \) and \( PO \) is stated in the following lemma, provided the bytecode program \( \hat{p} \) is the result of compiling the source program \( P \).

**Proposition 10** For every subprogram \( c \) of \( P \), proof obligations corresponding to \( c \) are equal to the proof obligations in \( \hat{p} \) that correspond to the subsequence \([c]\).
Lemma 11 For every label \( l \) in the program \( \hat{p} \):

\[
\text{wp}(l, \text{annot}, \text{post}, \chi) \land \bar{\gamma}_a(\text{loc}(l)) \Rightarrow \text{wp}(l, \text{annot}, \text{post}, \chi)
\]

provided the relation \( \models \subseteq \mathbb{D} \times \text{Guard} \) is valid, and the analysis \( \text{loc} \) is verifiable.

The hybrid VCGen follows from soundness of the standard VCGen. In addition, soundness of the VCGen for source programs follows if the compiler is semantics preserving.

Given a specification \( \text{pre}, \text{annot}, \text{post}, \chi \) for the bytecode program \( \hat{p} \), a non-hybrid VCGen extracts the set of proof obligations \( \text{po} \cup \{ \text{pre} \Rightarrow \text{wp}(0)[\gamma_{\text{code}}] \} \), where \( \text{wp} \) and \( \text{po} \) are defined in Figure 10. To avoid ambiguity, in the sequel we make explicit some parameters needed in the definition of \( \text{wp} \), \( \text{wp} \), \( \text{wp} \), and \( \text{wp} \). We write for instance \( \text{wp}(l, \text{annot}, \text{post}, \chi) \) instead of simply \( \text{wp}(l) \).

Let \( \text{loc} \) be a result of the analysis for the bytecode program \( \hat{p} \). Consider the specifications \( \{ \text{pre}, \text{annot}, \text{post}, \chi \} \) and \( \{ \text{pre}, \text{annot}, \text{post}, \chi \} \) for program \( \hat{p} \), such that for all \( l \in \text{dom}(\text{annot}) \), \( \text{annot}(l) \) is defined as \( \text{annot}(l) \land \bar{\gamma}_a(\text{loc}(l)) \).

We say that the relation \( \models \subseteq \mathbb{D} \times \text{Guard} \) is valid if for every abstract element \( d \in \mathbb{D} \) and \( b \in \text{Guard} \) we have that \( d \models b \) implies the universal validity of \( \bar{\gamma}_a(d) \Rightarrow b \). The result of the analysis \( \text{loc} \) is said verifiable if the set of proof obligations \( \text{po}(true, \bar{\gamma}_a \circ \text{loc}, true, true) \) are provable.

**Figure 6. Definition of WP function**

The soundness of the VCGen \( \text{po} \) follows from the following result and the soundness of the standard VCGen \( \text{po} \):

**Proposition 12** The provability of the set of proof obligations \( \text{po}(\text{pre}, \text{annot}, \text{post}, \chi) \) follows from the provability of \( \text{po}(\text{pre}, \text{annot}, \text{post}, \chi) \).

Consider for instance the sequence of bytecode in Figure 9. Recall that \( \text{annot} \) is defined as \( A[i - 1] = A[0] \) and \( A[|A| - 1] = A[0] \) in \( k \) and \( k' \) respectively. Let \( \text{annot} \) be defined by strengthening \( \text{annot} \) with the result of the analysis, i.e. \( \text{annot}(k) = \text{annot}(k) \land 0 \leq i \leq |A| \text{ (we can let } \text{annot}(k') = \text{annot}(k') \) ). Let \( \Psi \) be the weakest precondition computed by the non hybrid VCGen at label \( k_1 \):

\[
0 \leq i < |A| \Rightarrow (\text{upd}(A, i, A[0])[i + 1 - 1] = A[0]) \\
\land 0 \leq i + 1 \leq |A|
\]

\[
\Rightarrow \text{false}
\]

which, from Lemma 11 is implied by the hybrid wp and the
result of the analysis, i.e. by

\[
\text{upd}(A, i, A[0])|_{i + 1 - 1} = A[0] \land 0 \leq i < |A|
\]

As stated in Proposition 12, if the proof obligations returned by the hybrid VCgen are valid, and assuming the analysis is verifiable, we have that

\[
A[i - 1] = A[0] \land 0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow \text{upd}(A, i, A[0])|_{i + 1 - 1} = A[0]
\]

and \(0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow 0 \leq i < |A|\) are provable. Then, it follows that the verification condition

\[
A[i - 1] = A[0] \land 0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow \Psi
\]

returned by the standard VCgen is provable.

The above results establish that hybrid verification methods can be mapped to standard verification methods. In the context of PCC, one would like to establish the stronger result that hybrid certificates can be compiled into standard certificates. It is in fact possible to prove such a result, using the framework of \([4]\). However, the compilation of hybrid certificates into standard certificates requires using a certifying analyzer, that generates automatically logical proofs of correctness of the results of the analysis. While it is possible to avoid hybrid methods altogether, e.g. to rely on standard PCC architectures, hybrid methods are beneficial both for the code producer because they significantly reduce the number of proof obligations required to certify code, and for the code consumer, because they yield certificates that are more compact and more efficient to check. Certificate Translation from a hybrid to a standard VCgen is interesting for PCC scenarios in which original proof obligations are generated by a hybrid VCgen, but in which the targeted Trusted Computed Base has no support for hybrid certificates.

7 Related work

Some authors have formalized and proved the soundness of hybrid verification methods. For example, Wildmoser, Chaieb and Nipkow \([19]\) have used Isabelle/HOL to prove the soundness of hybrid methods for Java bytecode; they rely on interval analyses to detect arrays out of bounds, and implement a proof-producing version of the analysis that generates proofs that the results of the analysis is correct. More recently, Grégoire and Sacchini \([8]\) have formalized in Coq a hybrid verification method for JVM programs. They focus on a null-pointer analysis; although the analyzer is not formalized in Coq, Hubert and co-workers \([11]\) provides a good starting point for carrying such an implementation. The method of Grégoire and Sacchini \([8]\) supports bidirectional interaction between the analysis and verification condition generation, as the static analysis can extract useful information from the program annotations, at the same time as the results of the analysis are exploited by the verification condition generator to reduce the number of proof obligations (although we have not done so, it is relatively easy to integrate such bidirectional interaction in our work).

It is folklore that deductive verification methods can be viewed as abstract interpretation \([7, 6]\). Logical abstract interpretation \([9]\) explores the interaction between analysis and verification from the perspective of using theorem proving to improve the precision of abstract interpretations, and combinations of them.

Finally, the paper is closely related to previous works on proof-transforming compilation, and proof-producing program analyses. Saabas and Uustalu \([17]\) provide an algorithm to transform proofs in Hoare logic into proofs in compositional proof systems for assembly programs. Moving to more realistic languages, Müller et al. define proof-transforming compilation for Java and Eiffel \([1, 14, 16]\).
kow [19] in the context of verification condition generation for a bytecode language and Seo, Yang and Yi [18] in the context of a Hoare logic for a simple imperative language.

8 Conclusion

Program verification environments increasingly rely on hybrid methods to prove software correctness. In this paper, we have shown that hybrid verification methods at source and bytecode levels are tightly related, and that hybrid verification methods can be “compiled” into standard verification methods, which ensure that hybrid methods are sound.

Our next goal is to extend our results to more realistic languages and analyses. Modern languages include features, e.g. exceptions, that can potentially yield very large control flow graphs, making hybrid certificates particularly necessary; we expect that our results on preservation of proof obligations for Java [3] will scale to hybrid methods.

Besides, it would be beneficial to allow hybrid methods to rely on more advanced static analyses that provide valuable information for proving properties of programs. We intend to focus on the recent analysis of [10], and to develop a hybrid verification method based on this analysis.

References


