Synthesis of Circular Compositional Program Proofs via Abduction ⋆

Boyang Li[1], Isil Dillig[1], Thomas Dillig[1], Ken McMillan[2], and Mooly Sagiv[3]

[1] College of William & Mary
[2] Microsoft Research
[3] Tel Aviv University

Abstract. This paper presents a technique for synthesizing circular compositional proofs of program correctness. Our technique uses abductive inference to decompose the proof into small lemmas, which are represented as small program fragments annotated with pre and post-conditions. Different tools are used to discharge each different lemma, combining the strengths of different verifiers. Furthermore, each lemma concerns the correctness of small syntactic fragments of the program, addressing scalability concerns. We have implemented this technique and used it combine four different verification tools. Our experiments show that our technique can be successfully used to verify applications that cannot be verified by any individual technique.

1 Introduction

Different program verifiers have different limitations. For example, some may fail to prove a property because they use a coarse abstraction of the program semantics. In this category, we find abstract interpreters and verification condition generators, which require the property to be proved to be inductive. Others model the program semantics precisely, but often do not scale well in practice. In this category, we find model checkers and inductive invariant generators. To accomodate the limitations of program verifiers, a classical approach is synthesizing *compositional proofs*. The idea is to decompose the correctness proof of the program into a collection of lemmas, each of which can be verified by considering a small syntactic fragment of the program. This directly addresses the question of scalability, and indirectly the question of abstraction, since each lemma may be provable using a fairly coarse abstraction, even if the overall property is not.

The key difficulty in synthesizing compositional proofs is to discover a suitable collection of lemmas. Automating this process has proven to be extremely challenging. Some progress has been made in the finite state case [1, 2] and in some particular domains such as shape analysis [3]. However, general approaches for inferring compositional proofs are lacking.

In this paper, we describe an approach to inferring lemmas based on logical abduction, the process of inferring premises that imply observed facts. Specifically, our technique uses abduction to synthesize *circular compositional proofs*.

⋆ This work is supported in part by DARPA #FA8750-12-2-0020

```
1.    int i=1; int j=0;
2.    while(*) { j++; i+=3; }
3.    int z=i-j;
4.    int x=0; int y=0; int w=0;
5.    while(*) [assert(x=y)]
6.    { z+=x+y+w; y++; x+=z%2; w+=2; }
```

Fig. 1. Example to illustrate main ideas of our technique

In such a proof, each lemma is a fact that must hold at all times, and we must prove that each lemma is not the first to fail. In effect, the proof of each lemma is allowed to assume the correctness of all the others, the apparent circularity being broken by induction over time. Our goal is to introduce lemmas that can be discharged in this way, using only small program fragments.

A key feature of our approach is that it is lazy. That is, when a lemma $\mathcal{L}$ cannot be discharged, our technique introduces a new lemma that may help to prove $\mathcal{L}$. The key insight is that such useful auxiliary lemmas can be inferred by combining verification condition (VC) generation with logical abduction [4]. Specifically, given an invalid VC $\phi_1 \Rightarrow \phi_2$, we employ abductive inference to infer an auxiliary lemma ψ such that $\psi \wedge \phi_1 \Rightarrow \phi_2$ is valid. Experimentally, we observe that lemmas generated to help verification condition checking are also useful for other types of verifiers, such as model checkers and abstract interpreters.

The ability to synthesize compositional proofs by inferring relevant lemmas has two important benefits. First, it helps us to address the problems of scale and abstraction. The lemmas can be verified on small program fragments, and each can be checked using a different abstraction. Second, lemmas allow us to combine the strengths of many verifiers, as each lemma may be verified by a different tool. The tools can be used as black boxes, without any modification.

This paper applies these ideas for verifying safety properties of sequential programs. In principle, though, they can be applied to any class of programs and any proof system generating verification conditions in a suitable form.

1.1 Overview

Given an imperative program containing *assume* and *assert* statements, we want to show that no assertion fails in any execution. Our safety proof makes use of two basic steps: introduction and elimination of assertions. In an introduction step, we insert a new assertion at any point in the program. In an elimination step, we prove that some assertion always holds and then convert it to an assumption. When verifying an assertion $\mathcal{A}$, we can convert all the other assertions to assumptions, since we are only proving that $\mathcal{A}$ is not the first to fail. Moreover, given these assumptions, we might be able to verify our assertion locally, using some small fragment of the program containing the assertion.

As an example, consider the program of Figure 1. The assertion in square brackets on line 5 represents an *invariant* of the loop. It must hold each time the loop is entered and also when the loop exits. We would like to verify this invariant assertion using just lines 4–6 in isolation. This is not possible, however, because we require the precondition "z is odd" established by lines 1–3. Having failed in our verification attempt, we will try to infer a lemma that makes the verification possible. For this, we decorate the program with symbols representing unknown assumptions. We then compute a *verification condition* (VC), that is, a logical formula whose validity implies the correctness of the decorated program. Then, using a technique known as *abduction*, we will solve for values of the unknown assumptions making the VC valid. These assumptions will then become lemmas to be proved. Going back to our example, we decorate lines 4–6 as follows:

```
4.    int x=0; int y=0; int w=0;
      assume $\phi_1$
5.    while(*) [assert x=y; assume $\phi_2$]
6.          z+=x+y+w; y++; x+=z%2; w+=2;
```

The symbols ϕ_1 and ϕ_2 are placeholders for unknown assumptions. The assumption ϕ_1 is a precondition for the loop, while ϕ_2 is an additional (assumed) invariant. Our VC generator tells us that our decorated program is correct when the following formulas are valid:

$$(z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge \phi_1) \Rightarrow x = y$$
$$(\phi_2 \wedge x = y) \Rightarrow \mathrm{wp}(\sigma, x = y)$$

Here, σ is the loop body (the code of line 6), and $\mathrm{wp}(\sigma, \phi)$ stands for the weakest liberal precondition of formula ϕ with respect to statement σ. These conditions say that the invariant $x = y$ must hold on entering the loop, and that it is preserved by the loop body, given our assumptions.

Now, we can easily see that the first condition is valid, but the second one is not valid. Using the definition of wp, the second condition is equivalent to:

$$(\phi_2 \wedge x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

To prove the invariant $x = y$, we need to find a formula to plug in for ϕ_2 that makes this formula valid. At the same time, we do not want our new lemma ϕ_2 to contradict the original lemma $x = y$ that we are trying to prove. Thus, we want $\phi_2 \wedge x = y$ to be satisfiable. This problem of inferring a hypothesis that implies some desired fact, while remaining consistent with given facts, is known as *abduction*. Using the algorithm described in Section 4, we obtain the solution $(w + z)\%2 = 1$ for this abduction problem.

Having inferred an auxiliary invariant $(w+z)\%2 = 1$ through abduction, this formula now becomes a lemma in our proof. We *introduce* the invariant assertion "assert $(w + z)\%2 = 1$", so lines 4–6 now look like this:

```
4.    int x=0; int y=0; int w=0;
5.    while(*) [assert x=y; assert (w+z)%2 = 1]
6.          z+=x+y+w; y++; x+=z%2; w+=2;
```

We can now prove the assertion $x = y$ by assuming our new lemma. We therefore *eliminate* this assertion by converting it to an assumption, obtaining:

```
4.    int x=0; int y=0; int w=0;
5.    while(*) [assume x=y; assert (w+z)%2 = 1]
6.          z+=x+y+w; y++; x+=z%2; w+=2;
```

Unfortunately, the lemma $(w + z)\%2 = 1$ still cannot be proved using just these code lines, since it depends on the initial value of z, which is determined by the first loop. Therefore, we once again decorate the program with unknown assumptions ϕ_1 and ϕ_2. The VC's of the new program are:

$$(z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge \phi_1 \wedge x = y) \Rightarrow (w + z)\%2 = 1$$
$$\phi_2 \wedge (w + z)\%2 = 1 \wedge x = y \Rightarrow \mathrm{wp}(\sigma, x = y \Rightarrow (w + z)\%2 = 1)$$

where again σ is the loop body. That is, our lemma must hold on entry to the loop, and must be preserved by the loop, given our assumptions. However, neither of these conditions is valid, so we try to repair the first condition. To make it valid, we need to find a formula ψ to plug in for ϕ_1 such that:

$$(\psi \wedge z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge x = y) \Rightarrow (w + z)\%2 = 1$$
$$(\psi \wedge z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge x = y) \not\Rightarrow \mathit{false}$$

That is, the assumption ψ must be sufficient to establish the invariant on entry to the loop, but not contradict known facts, including the invariant $x = y$. Our abduction technique discovers the solution $z\%2 = 1$ for ψ.

This solution $z\%2 = 1$ for ϕ_1 now becomes a lemma, introduced as an assertion before the loop. We now have:

```
4.    int x=0; int y=0; int w=0;
      assert z%2 = 1;
5.    while(*) [assume x=y; assert (w+z)%2 = 1]
6.          z+=x+y+w; y++; x+=z%2; w+=2;
```

At this point we have two assertions in the program. The VC for the loop invariant is still not valid (that is, the invariant is not inductive). However, at this point we *can* verify it using just lines 4–6 in isolation, since we have the necessary precondition $z\%2 = 1$. Converting this assertion to an assumption, we give the above fragment to a client program analyzer. If this client tool is able to infer divisibility facts, it can verify the invariant by inferring the auxiliary invariant $w\%2 = 0$. We have therefore *localized* the verification of the loop invariant.

Having verified the assertion $(w + z)\%2 = 1$, we eliminate it by converting it to an assumption and we move on to the remaining assertion, $z\%2 = 1$. This assertion can be verified using lines 1–4 in isolation. That is, we give these lines to a client program analyzer that is able to infer the linear invariant $i = 3j + 1$ of the first loop. From this, it can prove that z is odd. All assertions have now been eliminated, so the program is verified.

Notice that our inference of lemmas using abduction had two significant advantages in this example. First, it allowed us to *localize* the verification, proving one lemma using just the first loop, another one using just the second. This addresses the issue of scale. Second, we were able to verify these lemmas using two different *abstractions*, in one case using divisibility predicates, and the other using linear equalities. In this way, proof decomposition allows different program verification tools to be combined as black boxes.

2 Language and Preliminaries

In this section, we give a small language on which we formalize our technique:

$$\begin{array}{ll}
\text{Program } Pr & := s \\
\text{Statement } s & := \texttt{skip} \mid v := e \mid s_1; s_2 \mid \texttt{if}(\star)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & \mid \texttt{while}(\star)[s_1]\ \texttt{do}\ \{s_2\} \mid \texttt{assert}\ p \mid \texttt{assume}\ p \\
\text{Expression } e & := v \mid c \mid e_1 + e_2 \mid e\%c \mid c * e \\
\text{Predicate } p & := e_1 \oslash e_2 \quad (\oslash \in \{<, >, =\}) \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p
\end{array}$$

A program consists of one or more statements. Statements include skip, assignments, sequencing, if statements, while loops, assertions, and assumptions. While loops may be decorated with invariants using the $[s]$ notation. The code s is executed before the loop body and also before exiting the loop, and may contain assert and assume statements. Expressions include variables, constants, addition, multiplication, and mod expressions. Predicates are comparisons between expressions as well as conjunction, disjunction, and negation.

We assume a scheme for numbering the statements in a program, including compound statements. Given a program π and a statement number (or *position*) p occurring in π, we write $\pi|_p$ for the statement in π numbered p. Moreover, given a statement σ, we write $\pi[\sigma]_p$ for π with σ replacing the statement numbered p. We also use $\text{asrts}(\pi)$ to represent the set of positions of assert statements in π and $\text{elim}(\pi, P)$, where P is a set of assert positions, to represent π with all asserts in positions P converted to assumes. The notation $\text{elim}(\pi, \neg p)$ is a shorthand for $\text{elim}(\pi, \text{asrts}(\pi) \setminus \{p\})$, that is, π with all asserts *except* position p converted to assumes. We use $\text{elim}(\pi)$ for π with all asserts converted to assumes.

3 Searching for Circular Compositional Proofs

In our proofs, we use a vocabulary Σ_U of placeholder symbols to stand for unknown program invariants. A placeholder $\phi \in \Sigma_U$ may occur only in a statement of the form "assume ϕ". We also use an operator spr that, given a program π, returns a formula whose validity implies correctness of π. That is, $\models \text{spr}(\pi)$ implies $\models \text{wp}(\pi, \mathit{true})$. The operator spr is, in effect, our VC generator. We assume that our VC generator spr returns a set of clauses of the form:

$$\chi \wedge \phi_p \Rightarrow \Gamma$$

$$\frac{\vdash \pi[\text{assert } \psi;\, \sigma]_p}{\vdash \pi[\sigma]_p}\ \textsc{Intro} \qquad \frac{\begin{array}{l}\vdash \text{elim}(\pi, \neg p)\\ \vdash \text{elim}(\pi, p)\end{array}}{\vdash \pi}\ \textsc{Elim} \qquad \frac{\begin{array}{l}\vdash \sigma\\ \vdash \pi[\text{elim}(\sigma)]_p\end{array}}{\vdash \pi[\sigma]_p}\ \textsc{Localize}$$

Fig. 2. Inference rules for compositional proof

where $\phi_p \in \Sigma_U$. The *constraint* χ does not contain placeholders, and the *goal* Γ is some formula asserted in the program. We also allow placeholder-free clauses of the form $\chi \Rightarrow \Gamma$. Our VC generation scheme (Section 3.3) is designed to produce VC's in these forms.

Our proof search algorithm makes use of three proof rules shown in Figure 2. These rules produce judgements of form $\vdash \pi$, where π is a program. The meaning of this judgement is that π does not fail in any context, i.e., $\text{wp}(\pi, \mathit{true}) = \mathit{true}$.

Rule Intro allows us to insert a new assertion in any syntactic position in the program. This rule is sound because adding an assertion can only strengthen the weakest precondition. The Intro rule is used in our proof search algorithm to introduce auxiliary lemmas in the form of assertions in the source code.

Rule Elim allows us to eliminate an assertion that is true. It says that, if the program is correct with all assertions *except* p converted to assumes, then we can convert p to an assume. Effectively, the Elim proof rule justifies the use of circular compositional reasoning in our approach. This rule will be useful in our proof search algorithm because it says that we can assume the correctness of all other assertions in proving the correctness of assertion p.

Finally, the Localize rule allows us to syntactically localize the verification of an assertion. That is, if a fragment of the program containing assertion p is correct, then p is correct in the entire program. This rule allows us to decompose large programs into smaller syntactic components for verification. The leaf subgoal $\vdash \sigma$ in this rule will be discharged by an oracle, which is our set of program verifiers. If the oracle certifies that σ is correct, then we take $\vdash \sigma$ as an axiom.

In searching for a proof in this system, we must make a number of heuristic decisions. For example, we must decide in what order to process subgoals, and, at each subgoal, we must choose a proof rule to apply. When applying the Intro rule, we must choose where and what assertions to introduce. Similarly, for Elim, we must choose the order of elimination of assertions, and for Localize, we must decide what program fragment σ to use for the verification of an assertion. Moreover, if a subgoal is unprovable (for example, because we introduced an assertion that is not correct), then we require a backtracking strategy.

Our tactic for searching for a proof in this system is illustrated in pseudocode in Figure 3. To reduce clutter, we don't construct the actual proof. Instead we just return *true* if a proof of the goal $\vdash \pi$ is found. We start by choosing an arbitrary assertion p to eliminate using the Elim rule (line 3). We call procedure Localize (line 4) to produce a local fragment for verifying p, using the Localize rule. In our implementation we use the inner-most while loop σ containing p. We then ask the oracle to prove the assertion (including a VC check).

```
Procedure ProofSearch($\pi$):
    input: program $\pi$
    output: true if proof of $\pi$ succeeds

    (1)  let $P = \text{asrts}(\pi)$
    (2)  if $P$ is empty, return true
    (3)  choose some $p \in P$, and let $\pi' = \text{elim}(\pi, \neg p)$
    (4)  let $\sigma = \text{Localize}(\pi', p)$
    (5)  if the oracle certifies $\sigma$ or $\models \text{spr}(\pi')$ then
    (6)          return ProofSearch($\text{elim}(\pi, p)$)
    (7)  let $\mathcal{I} = \text{InferByAbduction}(\pi')$
    (8)  for each $(p', \phi)$ in $\mathcal{I}$ do
    (9)          let $\pi'' = \pi[\text{assert } \phi;\ \pi|_{p'}]_{p'}$
    (10)         if ProofSearch($\pi''$) then return true
    (11) done
    (12) return false
```

Fig. 3. Proof search algorithm.

If the oracle can prove σ, we move on to the remaining assertions by processing the second sub-goal of the Elim rule (line 6).

On the other hand, if the oracle fails, we use abduction to generate a sequence of possible lemma introductions in order to make p provable (line 7). We try these in turn, applying the Intro rule (line 9) and recurring on the generated subgoal (line 10). If this proof fails, we move on to the next lemma in the sequence, and so on, until the sequence is exhausted, at which point, we return failure.

3.1 Using Abduction to Infer New Assertions

The key step in our proof search algorithm is the InferByAbduction procedure, shown in Figure 4. This procedure takes a program π and suggests new assertions that may be introduced to help make π provable. The first step in this process is to decorate the program with some assumptions of the form "assume ϕ_p", where ϕ_p is a placeholder symbol corresponding to statement position p. These placeholders stand for possible assertions we could introduce in a compositional proof. We discuss the choice of the placeholder locations in Section 3.2.

The next step is to generate the VC for the decorated program using the spr operator (described in Section 3.3). This is a set of clauses of the form $\chi \Rightarrow \Gamma$ or $\chi \wedge \phi_p \Rightarrow \Gamma$. To prove the assertion, we need to choose values of the placeholders to make all of these implications valid. If there is an invalid clause of the form $\chi \Rightarrow \Gamma$ we cannot succeed, so we return the empty sequence. Otherwise, we consider each invalid clause of the form $\chi \wedge \phi_p \Rightarrow \Gamma$. We want to choose a formula to assign to ϕ_p in order to make the implication $\chi \wedge \phi_p \Rightarrow \Gamma$ valid. In addition, we do not want the implication to be vacuously true, thus, we require that $\chi \wedge \phi_p$ be consistent.

```
Procedure InferByAbduction($\pi$):
    input: program $\pi$
    output: lazy list of pairs $(p, \phi_p)$

    let $\pi' =$ Decorate($\pi$)
    let VC = spr($\pi$)
    if there exists an invalid clause $\chi \Rightarrow \Gamma$ in VC then return
    for each invalid clause $\chi \wedge \phi_p \Rightarrow \Gamma$ in VC do
        for each $\psi$ in Abduc($\chi, \Gamma$) do
            yield $(p, \psi)$
        done
    done
```

Fig. 4. Inferring assertions by abduction.

This leaves us with the following abduction problem. We must find a formula ψ over the program variables, such that the following two conditions hold:

$$\models \chi \wedge \psi \Rightarrow \Gamma \qquad \text{and} \qquad \not\models \chi \wedge \psi \Rightarrow \mathit{false}$$

In Section 4, we describe a method of solving this problem. For now, we assume a procedure Abduc that, given χ and Γ, returns a lazy list of solutions for ψ. InferByAbduction then returns the list of solutions $\psi_1, \psi_2, \ldots, \psi_n$ for each placeholder ϕ_p, paired with the corresponding program position p of ϕ_p.

3.2 Program Decoration

An important consideration in choosing the placement of placeholder assumptions is that each clause in the VC should contain a placeholder to allow us to to make progress when the VC is not valid (except, of course, for the whole program's precondition, which must be valid). In general, this placement strategy depends on the VC generation scheme. In our particular language and VC scheme, it suffices to put a placeholder at the head of each loop. To support localization (as seen in the example of Figure 1) we also add a placeholder before each loop. That is, the procedure Decorate replaces each statement of the form while($\star$)[σ]{τ} in a program with:

```
assume $\phi_{\text{pre}}$;
while($\star$)    [$\sigma$; assume $\phi_{\text{inv}}$] { $\tau$ }
```

As a heuristic matter, we consider introducing a precondition for a loop before introducing an invariant.

3.3 VC generation

The general approach we have described can use any VC generator function spr, provided the VC's can be rewritten into the required form. Here, we present a

$$(1)\frac{}{P, Q \vdash \texttt{skip} : \mathit{true}, P, Q} \qquad (2)\frac{Q' = \exists v'.(P[v'/v] \wedge v = (e[v'/v]))}{P, Q \vdash v := e : \mathit{true}, Q', Q[e/v]}$$

$$(3)\frac{Q' = P \wedge C \quad P' = Q \wedge C}{P, Q \vdash \texttt{assert } C : \mathit{true}, Q', P'} \qquad (4.1)\frac{\begin{array}{c} Q' = P \wedge C \quad P' = (C \Rightarrow Q) \\ C \text{ not placeholder} \end{array}}{P, Q \vdash \texttt{assume } C : \mathit{true}, Q', P'}$$

$$(4.2)\frac{\mathrm{VC}' = (P \wedge \phi_p(\boldsymbol{v}) \Rightarrow Q)}{P, Q \vdash \texttt{assume } \phi_p(\boldsymbol{v}) : \mathrm{VC}', \mathit{true}, \mathit{true}} \quad (5)\frac{\begin{array}{c} P, P' \vdash s_1 : \mathrm{VC}_1, Q', P'' \\ Q', Q \vdash s_2 : \mathrm{VC}_2, Q'', P' \end{array}}{P, Q \vdash s_1; s_2 : \mathrm{VC}_1 \wedge \mathrm{VC}_2, Q'', P''}$$

$$(6)\frac{\begin{array}{c} P, \mathit{true} \vdash I : \mathrm{VC}_1, \_, Q' \\ \mathit{true}, \mathit{true} \vdash \mathrm{elim}(I); s; I \ : \ \mathrm{VC}_2, \_, Q_2 \\ \mathit{true}, Q \vdash \mathrm{elim}(I) \ : \ \mathrm{VC}_3, P', Q_3 \\ \mathrm{VC}' = \mathrm{VC}_1 \wedge \mathrm{VC}_2 \wedge Q_2 \wedge \mathrm{VC}_3 \wedge Q_3 \end{array}}{P, Q \vdash \texttt{while}(\star)[I] \texttt{ do } \{s\} : \mathrm{VC}', P', Q'}$$

$$(7)\frac{\begin{array}{c} P, Q \vdash s_1 : \mathrm{VC}_1, Q_1, P_1 \qquad P, Q \vdash s_2 : \mathrm{VC}_2, Q_2, P_2 \\ Q' = Q_1 \vee Q_2 \qquad P' = P_1 \wedge P_2 \end{array}}{P, Q \vdash \texttt{if}(\star) \texttt{ then } s_1 \texttt{ else } s_2 : \mathrm{VC}_1 \wedge \mathrm{VC}_2, Q', P'}$$

Fig. 5. Rules describing computation of VC's

simple VC generation approach for programs without procedures that explicitly generates VC's in the form $\chi \wedge \phi_p \Rightarrow \Gamma$. The approach is based on propagating both strongest postconditions forwards and weakest preconditions backwards. However, we could also use a more standard approach based on just weakest preconditions with some rewriting of the result into the right form.

In our VC generation scheme, we generate a clause for each placeholder ϕ_p. Given the strongest postcondition of the code preceding p, this clause states that ϕ_p guarantees the weakest precondition of the code succeeding p. Since we can't compute preconditions and postconditions precisely for loops, we abstract these conditions, using the stated invariants of the loop. The result is a VC that is a sufficient but not necessary condition for the correctness of the program.

We describe our VC generation procedure as a set of inference rules (Figure 5) that produce judgements of the form $P, Q \vdash s \ : \ \mathrm{VC}', P', Q'$. The meaning of this judgement is that, if the environment of statement s guarantees precondition P and postcondition Q, then s will guarantee postcondition P' and precondition Q', given that VC' is valid. That is, the judgement is valid when $\models \mathrm{VC}'$ implies $\models P \Rightarrow \mathrm{wp}(s, P')$ and $\models Q' \Rightarrow \mathrm{wp}(s, Q)$.

For primitive statements s, we have $\mathrm{VC}' = \mathit{true}$, $P' = \mathrm{sp}(s, P)$ and $Q' = \mathrm{wp}(s, Q)$. Thus, our rules propagate strongest post-conditions forward and weakest pre-conditions backward. However, rule 4.2 is a special rule for placeholder assumptions. It produces a VC clause rather than propagating sp and wp.

For while loops (rule 6), we weaken the post-condition and strengthen the precondition by allowing entry to the loop in any state satisfying the stated loop invariants. The first premise guarantees that the loop invariant holds on entry, the second that the loop invariant is preserved by one iteration of the loop, and the third that exiting the loop satisfies its postcondition. One way to think of this is that, to verify a loop under pre- and post-conditions P and Q, we need to establish three Hoare triples: $\{P\}\ I\ \{\mathit{true}\}$ and $\{\mathit{true}\}\ \mathit{elim}(I); s; I\ \{\mathit{true}\}$ and $\{\mathit{true}\}\ \mathit{elim}(I)\ \{Q\}$. For example, in a typical case, we want to prove an invariant assertion ψ in a loop. The decorated loop looks like this:

$$\text{while}(\star) \quad [\text{assert } \psi;\ \text{assume } \phi_{\text{inv}}] \qquad \{\ s\ \}$$

According to the first premise of rule (6), the precondition Q' of the loop is the precondition of "assert ψ; assume ϕ_{inv}", which is ψ. The postcondition P' of the loop (third premise) is the postcondition of "assume ψ; assume ϕ_{inv}", which is *true*, since ϕ_{inv} is a placeholder. Finally, the second premise yields the VC from:

$$\text{assume } \psi;\ \text{assume } \phi_{\text{inv}};\ \ s;\ \ \text{assert } \psi;\ \text{assume } \phi_{\text{inv}};$$

This yields two clauses, one for each placeholder instance, according to rule 4.2. The first is $\psi \wedge \phi_{\text{inv}} \Rightarrow \text{wp}(s, \psi)$. The second is *true*. To make the VC valid, we need to find an assumption ϕ_{inv}, under which ψ is inductive. Furthermore, since we add an "assume ϕ_{pre}" statement before the loop, Rule (4.2) results in the generation of the VC clause $P \wedge \phi_{\text{pre}} \Rightarrow \psi$ where P is the precondition of ϕ_{pre}. Thus, to make this VC valid, we must find an appropriate solution for ϕ_{pre} that implies ψ holds initially. Finally, the third premise of Rule (6) results in the generation of the VC $\psi \wedge \phi_{\text{inv}} \Rightarrow Q$, meaning that we must find a strengthening ϕ_{inv} of ψ that implies loop postcondition Q.

For program π, our goal is to derive a judgement of the form $\mathit{true}, \mathit{true} \vdash \pi\ :\ \text{VC}', \_, Q'$. This judgement says that if VC$'$ is valid, then a sufficient condition for correctness of our program in any initial state is Q'. Thus, we have $\text{spr}(\pi) = \text{VC}' \wedge Q'$. Using our particular decoration scheme, we are guaranteed that each clause in VC$'$ has exactly one occurrence of a placeholder (rule 4.2), or is free of placeholders (other rules).

Finally, we note that propagating postconditions forward has an additional advantage for compositional verification. That is, when we pass a localized program loop to the oracle for verification, we can include the precondition for that loop computed by our VC generator as an additional constraint on the initial state. This can allow us to verify assertions with smaller localizations.

4 Performing Abductive Inference

We now describe our technique for performing abductive inference, which corresponds to the ABDUC function used in the INFERBYABDUCTION algorithm. Recall that, given formulas χ and Γ, abduction infers a formula ψ such that:

$$(1)\ \chi \wedge \psi \Rightarrow \Gamma \qquad (2)\ \text{SAT}(\chi \wedge \psi)$$

While there are many formulas ψ that satisfy these two conditions, a *useful* abductive solution in our setting should have two characteristics:

1. First, ψ should contain as few variables as possible because invariants typically describe relationships between a few key variables in the program. For example, if both $x = y$ and $x + 10z + 5w - 4k \leq 10$ are sufficient to explain Γ, it is preferable to start with the simpler candidate $x = y$.
2. Second, ψ should be as general (i.e., as logically weak) as possible. For example, if $x = 0 \wedge y = 0$ and $x = y$ are both solutions to the inference problem, we prefer $x = y$ because solutions that are too specific (i.e., logically strong) are unlikely to hold for all executions of the program.

To find solutions containing as few variables as possible, observe that $\chi \wedge \psi \Rightarrow \Gamma$ can be rewritten as $\psi \Rightarrow (\neg\chi \vee \Gamma)$. Now, consider a satisfying assignment σ of $\neg\chi \vee \Gamma$ consistent with χ. By definition of a satisfying assignment, $\sigma \Rightarrow (\neg\chi \vee \Gamma)$. Thus, any satisfying assignment of $\neg\chi \vee \Gamma$ consistent with χ is a solution for the abductive inference problem. However, since we are interested in solutions with as few variables as possible, we are not interested in full satisfying assignments of $\neg\chi \vee \Gamma$, but rather *partial* satisfying assignments. Intuitively, a partial satisfying assignment σ of φ assigns values to a subset of the free variables in φ, but is still sufficient to make φ true, i.e., $\sigma(\varphi) \equiv$ *true*. Therefore, to find an abductive solution containing as few variables as possible, we will compute a *minimum partial satisfying assignment* (MSA) of $\neg\chi \vee \Gamma$ [5]. An MSA of formula φ is simply a partial satisfying assignment of φ containing no more variables than other partial satisfying assignments of φ. Minimum satisfying assignments for many theories, including Presburger arithmetic used in this paper, can be computed using the algorithm described in [5].

Now, if an MSA of $\neg\chi \vee \Gamma$ contains a set of variables V, we know there exists an abductive solution containing only V. However, we want to find a logically weakest formula over V that still implies $\neg\chi \vee \Gamma$. It can be shown that a weakest formula over V that implies $\neg\chi \vee \Gamma$ is given by $\forall \overline{V}.\ (\neg\chi \vee \Gamma)$ where $\overline{V} = \text{Vars}(\neg\chi \vee \Gamma) - V$. Furthermore, since we typically prefer quantifier-free solutions, quantifier elimination can be used to eliminate $\overline{V}$ in theories that admit quantifier elimination (such as Presburger arithmetic used here).

Example 1. Consider the problem from Section 1.1 of finding a ψ such that:

$$(1)\ \psi \wedge P \wedge x = y \Rightarrow wp(S, x = y) \qquad (2)\ \text{SAT}(\psi \wedge P \wedge x = y) \qquad \text{where}$$

$$\begin{aligned} P &= (z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0) \\ wp(S, x = y) &= (x + (z + x + y + w)\%2 = y + 1) \end{aligned}$$

To solve this problem, we first compute an MSA of $x \neq y \vee \neg P \vee wp(S, x = y)$ consistent with $P \wedge x = y$. Using the algorithm of [5], an MSA is $z = 1, w = 0$. Since variables x, y, i, j are not in the MSA, we generate the formula $\forall x, y, i, j.\ x \neq y \vee wp(S, x = y)$. Using quantifier elimination, this formula is equivalent to $(z + w)\%2 = 1$, which is the abductive solution we used in Section 1.1.

Name	LOC	Time (s)	# queries	Polyhedra	Linear Cong	Blast	Compass	Provable by RP?
B1	45	0.6	2	✘	✘	✔	✘	✘
B2	37	0.2	2	✘	✔	✘	✘	✘
B3	51	1.0	2	✔	✘	✔	✘	✔
B4	59	0.4	3	✔	✘	✔	✘	✘
B5	89	0.6	3	✔	✘	✔	✘	✘
B6	60	0.5	5	✘	✔	✘	✔	✘
B7	56	0.6	2	✘	✘	✔	✔	✘
B8	45	0.2	2	✔	✘	✔	✘	✔
B9	59	0.5	1	✘	✘	✔	✘	✘
B10	47	0.2	2	✔	✘	✔	✔	✘

Fig. 6. Experimental results on micro benchmarks

4.1 Computing All Abductive Solutions

In the previous discussion, we described how to compute one solution to the abductive inference problem defined by χ and Γ. However, the InferByAbduction algorithm from Section 3 requires a lazy list of solutions. That is, given a set of previous solutions $\psi_1, \psi_2, \ldots, \psi_k$ for the abduction problem defined by χ and Γ, how do we compute a new solution ψ_{k+1} distinct from $\psi_1, \psi_2, \ldots, \psi_k$?

To find such a solution ϕ_{k+1}, we compute an MSA of $\neg\chi \vee \Gamma$, that is not only consistent with χ but also with the *negations* $\neg\psi_1$, $\neg\psi_2$, ..., $\neg\psi_k$ of each of the previous solutions. Given such an MSA containing variables V, the formula $\forall\overline{V}.\ (\neg\chi \vee \Gamma)$ yields a new solution distinct from previous solutions. The process terminates when there is no longer a consistent solution.

5 Implementation and Experimental Evaluation

We have implemented the proposed technique using the SAIL front-end [6] for C programs and the Mistral SMT solver [7, 5]. Mistral computes MSAs and performs quantifier elimination, which are necessary for performing abduction.

To evaluate our technique, we performed two experiments, one involving challenging synthetic benchmarks, and a second using open-source C programs. In both experiments, our oracle consists of four client tools: BLAST [8], the polyhedra abstract domain [9] implemented in the Interproc tool [10], the linear congruences domain [11] also implemented in Interproc, and Compass [12, 13].

The results of the first experiment are summarized in Figure 6. This experiment involves 10 synthetic benchmarks available from http://www.cs.wm.edu/~tdillig/tacas-benchmarks.tar.gz. None of these benchmarks can be verified using one of the four client tools alone. Furthermore, even if we conjoin the invariants inferred by each tool, the combined invariants are still not sufficient to prove the assertion. However, using the proposed technique, all ten benchmarks can be verified using BLAST, polyhedra, linear congruences, and Compass as clients.

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

Fig. 7. Experimental results on real benchmarks.

In Figure 6, the column labeled LOC shows the number of lines of code in each benchmark, and the column labeled "Time" shows analysis time in seconds, excluding the time taken by client tools to answer queries. The next column shows the number of queries our technique poses to clients. The next four columns show which of the analyses were able to successfully answer at least one query on a given benchmark. Finally, the last column shows whether the original benchmark can be verified using the reduced product [14] of the convex polyhedra and linear congruences abstract domains, as implemented in Interproc.

The main point of the first experiment is that all benchmarks from Figure 6 can be verified using the proposed technique, although no client tool can individually verify any benchmark. Furthermore, the number of queries to client tools is small, ranging from 1-5 queries. This indicates that our technique is able to home in on relevant lemmas necessary to localize the overall proof. Figure 6 also shows that it is often helpful to combine different approaches in the verification task. For example, BLAST and polyhedra were useful for verifying benchmark 3, whereas linear congruences and Compass were used to verify benchmark 6.

In a second experiment, summarized in Figure 7, we used the proposed technique for verifying assertions in real C programs. The programs we analyzed include a Linux device driver, an OpenSSH component, a coreutil application, and two modules from the GNU scientific library (GSL). These benchmarks range from 526 to 7233 lines of code. As in the previous experiment, none of these benchmarks can be verified by individual client tools alone (i.e., they either do not terminate or report a false alarm). However, when the four client tools are combined using our technique, all benchmarks can be successfully verified.

Figure 7 also shows that, although the original programs are quite large, the extracted program fragments provided to client tools are small, ranging in size from an average of 5 to 29 lines. This corroborates the claim that our technique often extracts subgoals on program fragments that are much smaller than the original program. Although analyses like the polyhedra domain do not typically work on programs of this size, our technique can utilize such expressive analyses in the verification task by extracting small proof subgoals.

6 Related Work

Compositional Verification The technique presented here is similar to other techniques for compositional verification such as [1, 2, 15]. Specifically, [1] and [2] use Angluin's L^* automata learning algorithm for learning assumptions in concurrent finite-state systems. In this work, we address synthesizing compositional proofs for sequential infinite-state systems, and our approach to generating missing assumptions is based on logical abduction rather than Angluin's learning algorithm. Similar to our proposed technique, the approach described in [15] also employs a circular compositional approach and uses different abstractions to discharge proof subgoals. However, in contrast to [15], our proof subgoals are generated automatically by abduction.

Combining Program Analyzers Most previous work on combining verification tools focuses on abstract interpretation. Specifically, the reduced cardinal product [14] and logical product [16] constructions allow combining different abstract domains. Our work differs from these approaches in several respects: First, we do not require client tools to be based on abstract interpretation and treat each client tool as a black box. Second, our technique is compositional and does not require client tools to verify the entire program, but instead proof subgoals represented as small code snippets. This aspect of our technique allows utilizing very expensive analyses even when verifying large programs. Third, unlike the reduced product construction, our technique is automatic and does not need to be reimplemented for combining different analyses.

The Hector tool described in [17] also allows information exchange between different analysis tools. However, Hector does not generate proof subgoals, and information exchange is through first-order logic rather than source code.

Use of Abduction in Verification Several other approaches have used abductive inference in the context of program verification [18, 3, 19]. Among these approaches, [3] and [19] also use abduction to generate missing preconditions. Specifically, [3] uses abduction for generating missing assumptions in an interprocedural shape analysis algorithm, whereas [19] uses abduction in the context of logic programming. Our work differs from [3, 19] in that we address combining different verification tools in a compositional way and use a different algorithm for computing abductive solutions. Our own recent work also uses abductive inference to semi-automate the task of classifying error reports as false alarms or real bugs [20]. Similar to [20], we use minimum satisfying assignments [5] to solve abductive inference problems. However, the present work addresses the very different problem of combining different verification tools in one framework.

7 Conclusion

We have proposed an algorithm for automatically synthesizing circular compositional proofs of program correctness. Our technique employs logical abduction to infer auxiliary lemmas that are useful in a compositional proof. The inference of helper lemmas allows combining the strengths of different program verifiers in one framework, as different verifiers can be used to discharge different lemmas. We have implemented the proposed technique, and our experiments show that it can verify programs that cannot be proven by individual tools.

8 Acknowledgments

We would like to thank Hongseok Yang, Aaron Bradley, Peter O'Hearn, Noam Rinetzky, and the anonymous reviewers for their helpful feedback.

References

1. Cobleigh, J., Giannakopoulou, D., Păsăreanu, C.: Learning assumptions for compositional verification. TACAS (2003) 331–346
2. Gupta, A., Mcmillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Form. Methods Syst. Des 2008
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL **44**(1) (2009) 289–300
4. Peirce, C.: Collected papers of Charles Sanders Peirce. Belknap Press (1932)
5. Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum satisfying assignments for SMT, CAV (2012)
6. Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language. Stanford University Technical Report
7. Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In: CAV. (2009)
8. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: International conference on Model checking software. (2003) 235–239
9. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: POPL, ACM (1978) 84–96
10. Jeannet, B.: Interproc analyzer for recursive programs with numerical variables. http://pop-art. inrialpes. fr/interproc/interprocweb. cgi
11. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: TAPSOFT'91, Springer (1991) 169–192
12. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: ESOP. (2010)
13. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. POPL (2011)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, ACM (1979) 269–282
15. McMillan, K.: Verification of infinite state systems by compositional model checking. Correct Hardware Design and Verification Methods (1999) 705–705
16. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: ACM SIGPLAN Notices. Volume 41., ACM (2006) 376–386
17. Charlton, N., Huth, M.: Hector: Software model checking with cooperating analysis plugins. In: Computer Aided Verification, Springer (2007) 168–172
18. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246
19. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proceedings of the 1994 International Symposium on Logic programming, Citeseer (1994) 377–391
20. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI. (2012)