Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-015-0397-7

TACAS 2013

Synthesis of circular compositional program proofs via abduction

Isil Dillig[1] · **Thomas Dillig**[1] · **Boyang Li**[2] · **Ken McMillan**[3] · **Mooly Sagiv**[4]

Abstract This paper presents a new technique for synthesizing circular compositional proofs of program correctness. Our technique uses abductive inference to decompose the proof into small lemmas (i.e., compositionality) and proves that each lemma is not the first one to fail (i.e., circularity). Our approach represents lemmas as small program fragments annotated with pre and post-conditions and uses different tools to discharge each different lemma. This approach allows us to combine the strengths of different verifiers and addresses scalability concerns, as each lemma concerns the correctness of small syntactic fragments of the program. We have implemented our proposed technique for generating circular compositional proofs and used four different program analysis tools to discharge the proof subgoals. We evaluate our approach on a collection of synthetic and real-world benchmarks and show that our technique can successfully verify applications that cannot be verified by any individual technique.

Keywords Program verification · Abductive inference · Circular compositional reasoning

This work is supported in part by NSF CAREER Award 1453386 and DARPA #FA8750-12-2-0020.

✉ Isil Dillig
isil@cs.utexas.edu

Thomas Dillig
tdillig@cs.utexas.edu

Boyang Li
bli01@email.wm.edu

Ken McMillan
kenmcmil@microsoft.com

Mooly Sagiv
msagiv@acm.org

[1] UT Austin, Austin, USA

[2] College of William and Mary, Williamsburg, USA

[3] Microsoft Research, Redmond, USA

[4] Tel Aviv University, Tel Aviv, Israel

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

Published online: 19 August 2015

abduction to synthesize circular compositional proofs. 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, although it is difficult to infer lemmas that might help abstract interpreters or model checkers prove a property, this problem is relatively straightforward for verification condition checkers. 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. The inference of such useful lemmas is a problem of logical abduction, a problem that has been studied in various contexts [3,4,20] and for which we have practical solutions [5]. 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.

Contributions To summarize, this paper makes the following contributions:

- We present new proof rules for synthesizing circular compositional proofs of program correctness.
- We show how *abductive inference* can be used to infer proof subgoals in the form of pre- and post-conditions of code snippets.
- We use our technique for combining the strengths of different program analyzers, as each lemma can be discharged by different program analysis tools.
- We experimentally evaluate our approach on a collection of synthetic and real-world benchmarks and show that our technique can be used to verify programs that cannot be verified by existing individual tools.

Outline of the paper We start by giving an overview of our approach in Sect. 2 and show how it decomposes the program's proof of correctness. We then give some preliminary definitions in Sect. 3 and introduce three key proof rules used in our safety proofs 4. Section 5 describes our proof search algorithm based on circular compositional reasoning, and Sect. 6 presents our abduction algorithm. The next two sections describe our implementation and experiments, with related work discussed in Sect. 9.

2 Overview

Given an imperative program containing *assume* and *assert* statements, our goal is to construct a safety proof that no assertion fails in any execution of this program. 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 because our goal is to prove that $\mathcal{A}$ is not the first assertion 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 Fig. 1. The assertion in square brackets on line 12 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 the second loop in isolation (lines 9–18). This is not possible, however, because we require the precondition "z is odd" established by lines 1–8. 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*,

Fig. 1 Example to illustrate main ideas of our technique

```
1.  int i=1;
2.  int j=0;
3.  while(*)
4.  {
5.    j++;
6.    i+=3;
7.  }
8.  int z = i-j;
9.  int x=0;
10. int y=0;
11. int w=0;
12. while(*) [assert(x=y)]
13. {
14.   z+=x+y+w;
15.   y++;
16.   x+=z%2;
17.   w+=2;
18. }
```

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 9–18 as follows:

```
9.    int x=0;
10.   int y=0;
11.   int w=0;
      assume φ1
12.   while(*) [assert(x=y) assume φ2 ]
13.   {
14.     z+=x+y+w;
15.     y++;
16.     x+=z%2;
17.     w+=2;
18.   }
```

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:

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

Here, σ is the loop body (the code in lines 14–17), and $\text{wp}(\sigma, \phi)$ 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 even when ϕ_1 is set to true, 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 Sect. 6, 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 9–18 now look like this:

```
9.    int x=0;
10.   int y=0;
11.   int w=0;
12.   while(*) [assert(x=y); assert((w+z)%2 = 1) ]
13.   {
14.     z+=x+y+w;
15.     y++;
16.     x+=z%2;
17.     w+=2;
18.   }
```

We can now prove the assertion $x = y$ by assuming our new lemma $(w+z)\%2 = 1$. That is, $x = y$ is *inductive relative to* $(w+z)\%2 = 1$, meaning that we can prove the inductiveness of $x = y$ *assuming* our new lemma. We therefore *eliminate* this assertion by converting it to an assumption, and obtain the following code:

```
9.    int x=0;
10.   int y=0;
11.   int w=0;
12.   while(*) [assume(x=y); assert((w+z)%2 = 1) ]
13.   {
14.     z+=x+y+w;
15.     y++;
16.     x+=z%2;
17.     w+=2;
18.   }
```

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:

```
      ...
      assume φ1
12.   while(*) [assume(x=y); assert((w+z)%2=1);
      assume φ2 ]
13.   {
14.     z+=x+y+w;
15.     y++;
16.     x+=z%2;
17.     w+=2;
18.   }
```

The VCs of the new program are:

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

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 when ϕ_1 and ϕ_2 are set to true, 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:

```
9.    int x=0;
10.   int y=0;
11.   int w=0;
      assert (z%2 = 1)
12.   while(*) [assume(x=y); assert((w+z)%2 = 1) ]
13.   {
14.     z+=x+y+w;
15.     y++;
16.     x+=z%2;
17.     w+=2;
18.   }
```

At this point we have two assertions in the program. The VC for the loop invariant is still not valid (that is, the lemma $(w + z)\%2 = 1$ is not inductive). However, at this point, it is possible to prove the invariant using just lines 9–18 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 1–8 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.

As this example also illustrates, our strategy for proving program safety is similar to an SMT solver. In lazy SMT, a SAT solver generates proof subgoals in the form of propositional assignments that must be verified by cooperating theory solvers. In our method, the core analysis consisting of VC generation and abduction generates proof subgoals in the form of program fragments to be verified by the client analyses. In SMT, the subgoals are combined by propositional resolution, whereas here they are combined by circular compositional reasoning. In both cases, the core prover is guided in its search by facts proved by the clients.

3 Language and preliminaries

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

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

In this language, 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 only 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. To avoid redundancy, we only allow non-deterministic conditions (i.e., $\star$) in loops and conditionals. However, observe that any condition can be expressed using assume statements in the body of the `then` and `else` branches as well as inside and after the loop body.

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 $asrts(\pi)$ to represent the set of positions of assert statements in π and $elim(\pi, P)$, where P is a set of assert positions, to represent π with all asserts in positions P converted to assumes. The notation $elim(\pi, \neg p)$ is a shorthand for $elim(\pi, asrts(\pi) \setminus \{p\})$, that is, π with all asserts *except* position p converted to assumes. We use $elim(\pi)$ for π with all asserts converted to assumes.

We give our programs a standard weakest-precondition semantics, defined as follows, by induction on the program structure:

$$\begin{aligned}
\mathrm{wp}(\mathrm{assert}\,\psi, \phi) &= \psi \wedge \phi \\
\mathrm{wp}(\mathrm{assume}\,\psi, \phi) &= \psi \Rightarrow \phi \\
\mathrm{wp}(x := e, \phi) &= \phi\langle e/x\rangle \\
\mathrm{wp}(s_1; s_2, \phi) &= \mathrm{wp}(s_1, \mathrm{wp}(s_2, \phi)) \\
\mathrm{wp}(\mathrm{if}(*)\mathrm{then}\, s_1 \,\mathrm{else}\, s_2, \phi) &= \mathrm{wp}(s_1, \phi) \wedge \mathrm{wp}(s_2, \phi) \\
\mathrm{wp}(\mathrm{while}(*)[t] do\{s\}, \phi) &= \wedge_{i\geq 0}\mathrm{wp}(s^i; t, \phi)
\end{aligned}$$

where s^i stands for a sequential composition $s; \ldots; s$ of i occurrences of s.

The meaning of a judgement $\vdash \pi$, where π is a program is that π does not fail internally in any environment, that is, $\mathrm{wp}(\pi, \mathit{true}) = \mathit{true}$.

The following lemma says that we can break the correctness of a program fragment π in a given context into two parts: verifying that π does not fail internally, and assuming it does not, that π satisfies its post-condition:

Lemma 1 *For all programs* π, $\mathrm{wp}(\pi, \phi) = \mathrm{wp}(\pi, \mathrm{true}) \wedge \mathrm{wp}(\mathit{elim}(\pi), \phi)$.

Proof By induction on the structure of programs. The interesting cases are assertions and sequential composition. First, suppose $\pi = \mathrm{assert}\psi$. Thus, $\mathrm{wp}(\pi, \phi) = \psi \wedge \phi$ and $\mathrm{wp}(\pi, \mathit{true}) = \psi$. Moreover, we have $\mathit{elim}(\pi) = \mathrm{assume}\psi$, so $\mathrm{wp}(\mathit{elim}(\pi), \phi) = \psi \Rightarrow \phi$. Thus, $\mathrm{wp}(\pi, \mathit{true}) \wedge \mathrm{wp}(\mathit{elim}(\pi), \phi) = \psi \wedge (\psi \Rightarrow \phi) = \psi \wedge \phi = \mathrm{wp}(\pi, \phi)$. Now suppose $\pi = s_1; s_2$. We have $\mathrm{wp}(\pi, \phi) = \mathrm{wp}(s_1, \mathrm{wp}(s_2, \phi))$. By inductive hypothesis, we then have

$$\begin{aligned}
\mathrm{wp}(\pi, \phi) &= \mathrm{wp}(s_1, (\mathrm{wp}(s_2, \mathit{true}) \wedge \mathrm{wp}(\mathit{elim}(s_2), \phi))) \\
&= \mathrm{wp}(s_1, \mathit{true}) \wedge \mathrm{wp}(\mathit{elim}(s_1), (\mathrm{wp}(s_2, \mathit{true}) \\
&\quad \wedge \mathrm{wp}(\mathit{elim}(s_2), \phi))) \\
&= \mathrm{wp}(s_1, \mathit{true}) \wedge \mathrm{wp}(\mathit{elim}(s_1), \mathrm{wp}(s_2, \mathit{true})) \\
&\quad \wedge \mathrm{wp}(\mathit{elim}(s_1), \mathrm{wp}(\mathit{elim}(s_2), \phi)) \\
&= \mathrm{wp}(s_1, \mathrm{wp}(s_2, \mathit{true})) \wedge \mathrm{wp}(\mathit{elim}(s_1); \mathit{elim}(s_2), \phi) \\
&= \mathrm{wp}(s_1; s_2, \mathit{true}) \wedge \mathrm{wp}(\mathit{elim}(s_1; s_2), \phi)
\end{aligned}$$

Note in the first and second lines above, we use the inductive hypothesis in the forward direction, while in the fourth we use it in the reverse direction. The remaining cases are straightforward. □

4 Proof rules

In this section, we introduce the three main proof rules, called INTRO, ELIM, and LOCALIZE underlying our technique. In the remainder of the paper, 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 the correctness of π. That is, $\models \mathrm{spr}(\pi)$ implies $\models \mathrm{wp}(\pi, \mathit{true})$. The operator spr is, in effect, our VC generator. We assume that our VC generator spr returns a set of clauses of the form:

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

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 (Sect. 5.3) is designed to produce VCs in these forms.

4.1 Proof rule INTRO

Our first proof rule, called INTRO, allows us to insert a new assertion in any syntactic position in the program:

$$\text{INTRO:}\quad \frac{\vdash \pi[\mathrm{assert}\,\psi; \sigma]_p}{\vdash \pi[\sigma]_p} \tag{1}$$

That is, if we can verify program π with any assertion ψ added in any arbitrary position of π, this implies that program π must be safe.

Theorem 1 *Rule* INTRO *is sound.*

Proof We observe that $\mathrm{wp}(\mathrm{assert}\,\phi; s, \psi) \Rightarrow \mathrm{wp}(s, \psi)$. That is, prefixing a statement with an assertion only makes the weakest precondition stronger. Thus, by using monotonicity of the wp rules, we have:

$$\mathrm{wp}(\pi[\mathrm{assert}\,\phi; s]_p, \psi) \Rightarrow \mathrm{wp}(\pi[s]_p, \psi)$$

□

4.2 Proof rule ELIM

Our second proof rule, called ELIM, allows us to eliminate an assertion that has been verified:

$$\text{ELIM:}\quad \frac{\vdash \mathit{elim}(\pi, \neg p) \qquad \vdash \mathit{elim}(\pi, p)}{\vdash \pi} \tag{2}$$

In particular, this rule 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 constructing our safety proof because it says that we can assume the correctness of all other assertions in proving the correctness of assertion p.

Theorem 2 *Rule* ELIM *is sound.*

Proof We show by induction on the structure of programs that:

$$\text{wp}(\pi, \phi) = \text{wp}(elim(\pi, \neg p), \phi) \wedge \text{wp}(elim(\pi, p), \phi)$$

Again, the interesting cases are assertions and sequential composition. First, suppose $\pi = \text{assert}\,\psi$. Assume first that the position of π is p. Then, $elim(\pi, \neg p) = \text{assert}\,\psi$ and $elim(\pi, p) = \text{assume}\,\psi$. This gives us

$$\begin{aligned}&\text{wp}(elim(\pi, \neg p), \phi) \wedge \text{wp}(elim(\pi, p), \phi)\\ &\quad = \phi \wedge \psi \wedge (\phi \Rightarrow \psi) = \phi \wedge \psi = \text{wp}(\pi, \phi).\end{aligned}$$

On the other hand, suppose the position of π is not p. Then, $elim(\pi, \neg p) = \text{assume}\,\psi$ and $elim(\pi, p) = \text{assert}\,\psi$, with the same result as above. Now suppose $\pi = s_1; s_2$. By Lemma 1, we have

$$\text{wp}(\pi, \phi) = \text{wp}(s_1, \mathit{true}) \wedge \text{wp}(elim(s_1), \text{wp}(s_2, \phi))$$

Now by two applications of the inductive hypothesis, distributing wp into conjunction, we have:

$$\text{wp}(\pi, \phi) = \begin{cases}\text{wp}(elim(s_1, \neg p), \mathit{true})\\ \wedge\text{wp}(elim(s_1, p), \mathit{true})\\ \wedge\text{wp}(elim(s_1), \text{wp}(elim(s_2, \neg p), \phi))\\ \wedge\text{wp}(elim(s_1), \text{wp}(elim(s_2, p), \phi))\end{cases}$$

Now since $elim(elim(s, p)) = elim(elim(s, \neg p)) = elim(s)$, we have, after reordering the conjunction:

$$\text{wp}(\pi, \phi) = \begin{cases}\text{wp}(elim(s_1, \neg p), \mathit{true})\\ \quad\wedge\text{wp}(elim(elim(s_1, \neg p)),\\ \quad\text{wp}(elim(s_2, \neg p), \phi))\\ \wedge\text{wp}(elim(s_1, p), \mathit{true})\\ \quad\wedge\text{wp}(elim(elim(s_1, p)),\\ \quad\text{wp}(elim(s_2, p), \phi))\end{cases}$$

So by Lemma 1 (in the reverse direction) we have:

$$\text{wp}(\pi, \phi) = \begin{cases}\text{wp}(elim(s_1; s_2, \neg p), \phi)\\ \wedge\text{wp}(elim(s_1; s_2, p), \phi)\end{cases}$$

The remaining cases are straightforward. □

4.3 Proof rule LOCALIZE

Our third and final proof rule, called *Localize* allows us to syntactically localize the verification of an assertion:

LOCALIZE:

$$\frac{\vdash \sigma \qquad \vdash \pi[elim(\sigma)]_p}{\vdash \pi[\sigma]_p} \tag{3}$$

According to this rule, 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.

Theorem 3 *Rule* LOCALIZE *is sound.*

Proof By Lemma 1, we have $\text{wp}(\sigma, \phi) = \text{wp}(\sigma, \mathit{true}) \wedge \text{wp}(elim(\sigma), \phi)$. By the first premise of the rule, we have $\text{wp}(\sigma, \mathit{true}) = \mathit{true}$. Thus $\text{wp}(\sigma, \phi) = \text{wp}(elim(\sigma), \phi)$, which implies $\text{wp}(\pi[\sigma]_p, \mathit{true}) = \text{wp}(\pi[elim(\sigma)]_p, \mathit{true})$. □

5 Algorithm for constructing circular compositional safety proofs

In this section, we describe our algorithm for constructing circular compositional safety proofs. While our algorithm is based on the three proof rules from the previous section, we must make a number of heuristic decisions in searching for a proof in this system. 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 pseudo-code in Fig. 2. 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.

Our algorithm starts by choosing an arbitrary assertion p to eliminate using the ELIM rule (line 3). That is, we first convert all assertions except p to assumes, and then try to prove p. In particular, we call procedure LOCALIZE (line 4) to produce a local fragment for verifying p, using the *Localize* rule. In our implementation we use the innermost while loop

```
Procedure ProofSearch(π):
    input: program π
    output: true if proof of π succeeds

(1)  let P = asrts(π)
(2)  if P is empty, return true
(3)  choose some p ∈ P, and let π' = elim(π, ¬p)
(4)  let σ = Localize(π', p)
(5)  if the oracle certifies σ or ⊨ spr(π') then
(6)      return ProofSearch(elim(π, p))
(7)  let I = InferByAbduction(π')
(8)  for each (p', φ) in I do
(9)      let π'' = π[assert φ; π|_{p'}]_{p'}
(10)     if ProofSearch(π'') then return true
(11) done
(12) return false
```

Fig. 2 Proof search algorithm

σ containing p. We then ask an oracle (i.e., a client program analysis tool) to prove the assertion. 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). That is, we now turn assertion p into an assume and recursively invoke ProofSearch in order to prove the remaining assertions.

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, meaning that we apply the Intro rule (line 9) to generate a new subgoal to prove and try to verify the program containing this additional lemma (lemma 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.

The remainder of this section explains the various auxiliary procedures used in our ProofSearch algorithm in more detail.

5.1 Using Abduction to Infer New Assertions

The key step in our proof search algorithm is the InferByAbduction procedure, shown in Fig. 3. 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 Sect. 5.2.

```
Procedure InferByAbduction(π):
    input: program π
    output: lazy list of pairs (p, φ_p)

    let π' = Decorate(π)
    let VC = spr(π')
    if there exists an invalid clause χ ⇒ Γ in VC then return
    for each invalid clause χ ∧ φ_p ⇒ Γ in VC do
        for each ψ in Abduc(χ, Γ) do
            yield (p, ψ)
        done
    done
```

Fig. 3 Inferring assertions by abduction

Having decorated the program with the appropriate assumptions, the next step is to generate the VC for the decorated program using the spr operator (described in Sect. 5.3). 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.

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:

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

In Sect. 6, 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.

5.2 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 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 Fig. 1) we also add a placeholder before each loop. That is, the procedure Decorate replaces each statement of the form while$(\star)[\sigma]\{\tau\}$ in a program with:

$$\begin{array}{l}\text{assume } \phi_{\text{pre}};\\ \text{while}(\star) \quad [\sigma; \text{assume } \phi_{\text{inv}}]\ \{\ \tau\ \}\end{array}$$

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

5.3 VC generation

The general approach we have described can use any VC generator function spr, provided the VCs can be rewritten into the required form. Here, we present a simple VC generation approach for programs without procedures that explicitly generates VCs 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 (Fig. 4) 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 $\{\mathit{true}\}\ elim(I); s; I\ \{\mathit{true}\}$ and $\{\mathit{true}\}\ elim(I)\ \{Q\}$. 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

$$(1)\frac{}{P, Q \vdash \texttt{skip} : \mathit{true}, P, Q}$$

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

$$(3)\frac{Q' = P \wedge C \quad P' = Q \wedge C}{P, Q \vdash \texttt{assert } C : \mathit{true}, Q', P'}$$

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

$$(4.2)\frac{\mathrm{VC}' = (P \wedge \phi_p(\mathbf{v}) \Rightarrow Q)}{P, Q \vdash \texttt{assume } \phi_p(\mathbf{v}) : \mathrm{VC}', \mathit{true}, \mathit{true}}$$

$$(5)\frac{\begin{array}{c}P, P' \vdash s_1 : \mathrm{VC}_1, Q', P'' \\ Q', Q \vdash s_2 : \mathrm{VC}_2, Q'', P'\end{array}}{P, Q \vdash s_1; s_2 : \mathrm{VC}_1 \wedge \mathrm{VC}_2, Q'', P''}$$

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

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

Fig. 4 Rules describing computation of VCs

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 $\psi \wedge \phi_{\text{inv}} \Rightarrow \mathrm{wp}(s, \psi)$. 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 $\mathit{true}, \mathit{true} \vdash \pi : \mathrm{VC}', \_, Q'$. 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 $\mathrm{spr}(\pi) = \mathrm{VC}' \wedge Q'$. 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.

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

Observe that one obvious solution to this problem is $\psi = \Gamma$. In the context of our algorithm, since ψ corresponds to a candidate fact we tried to prove but could not, this trivial solution corresponds to querying the same subgoal we tried before! Thus, to avoid obtaining this trivial and useless solution, our goal is to utilize what we already know (i.e., χ) to find a new subgoal ψ that, together with χ, is sufficient to establish Γ.

Hence, in our setting, we believe a useful abductive solution 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. Furthermore, since $x = 0 \wedge y = 0$ implies $x = y$, client analyses that can prove the stronger invariant should also be able to prove the weaker invariant $x = y$.

The second criterion of generality above has an obvious technical characterization. In particular, if ψ and ψ' are both abductive solutions and $\psi' \Rightarrow \psi$, this means ψ' is less general than ψ. Therefore, the most general explanation corresponds to the logically weakest solution. While there is no obvious technical definition of the first criterion of simplicity, we will characterize simplicity of an explanation ψ by the number of variables in ψ. We believe this characterization is sensible because most invariants involve relationships between a few key variables rather than every variable in the program. Thus, in our setting, the notion of generality is captured through logical implication, whereas simplicity is approximated by the number of variables involved in the invariant. We will first describe how to generate abductive explanations with as few variables as possible; then, we will consider the problem of finding the weakest solution involving a certain set of variables.

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., $\sigma(\varphi) \equiv \mathit{true}$. 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).

After computing a most general abductive solution ψ with the fewest number of variables as described above, we perform one more step to further simplify ψ. In particular, we do not want ψ to repeat facts that we already know. For example, suppose one of the invariants we already know is $z \geq 10$, and the abductive solution ψ is $x = y \wedge z > 0$. In this case, the subpart of the formula $z > 0$ is redundant since it is already implied by the known invariant $z \geq 10$. Therefore, to ensure that the subgoals we create do not repeat facts we have already shown, we simplify the abductive solution ψ with respect to Γ. In particular, subparts of ψ that are implied by Γ are replaced with *true*, while subformulas of ψ that contradict Γ are replaced with *false*. A more technical discussion of how this simplification is performed is given in [21].

Example 1 Consider the problem from Sect. 2 of finding a ψ such that:

(1) $\psi \wedge P \wedge x = y \Rightarrow wp(S, x = y)$

(2) $\text{SAT}(\psi \wedge P \wedge x = y)$ where

$$P = (z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0)$$
$$wp(S, x = y) = (x + (z + x + y + w)\%2 = y + 1)$$

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 Sect. 2.

6.1 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 Sect. 5 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* $\neg\psi_1, \neg\psi_2, \ldots, \neg\psi_k$ 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.

7 Implementation and extensions

We have implemented the techniques described in this paper using the SAIL front-end [6] for analyzing C programs. Our implementation also relies on the Mistral constraint solver [5,7] for solving and simplifying constraints generated by the analysis and for computing minimum satisfying assignments and performing quantifier elimination which are necessary for performing abductive inference as described in Sect. 6.

To simplify the key technical ideas, the ProofSearch procedure described in Sect. 5 reanalyzes the program from scratch on each recursive invocation. In our implementation, we only incrementally recompute those pre and post-conditions that may have changed.

While we formalize our technique on a small language without pointers, our implementation handles most features of the C language, including pointers, arrays, and function calls. To allow our technique to work on pointer-manipulating programs, we also integrate a flow-sensitive pointer analysis. In particular, for computing strongest postconditions and weakest preconditions in the presence of pointers, we utilize an environment Γ that maps each pointer variable to a set of locations it may point to. For example, suppose that we want to compute the weakest precondition of *true* with respect to the statement `assert(*p == 3)`, and suppose that $\Gamma(\texttt{p}) = \{\texttt{a}, \texttt{b}\}$. In this case, the weakest precondition is computed as:

$$(p = \&a \rightarrow a = 3) \wedge (p = \&b \rightarrow b = 3)$$

This precondition states that if `p` points to `a`, then `a` must have the value 3 before this statement; similarly, if `p` points to `b`, then `b` must have the value 3 before the statement. Effectively, our computation utilizes points-to information to capture pre- and postconditions of statements parametric over the available points-to facts.

For interprocedural analysis, our implementation is context-sensitive. Specifically, for computing strongest postconditions, a bottom-up analysis computes strongest postconditions of callees parametric over function inputs, which are then instantiated at calling contexts. While a bottom-up analysis may at first seem counterintuitive for a forwards analysis, this is necessary if we want to reuse the callee's postcondition summary in every calling context. As an example, consider the following function `foo` and its caller `bar`:

```
int x, y;
void bar() { x=0; y=0; foo(); }
void foo() { x=x+1; y=x; x=x+2; }
```

Here, the summary of `foo` is computed as $x = x_0 + 3 \wedge y = x_0 + 1$ where x_0 denotes the value of `x` at any call site of `foo`. At a call site of `foo`, let ϕ be the strongest postcondition right before the function call, and let φ be the callee's summary. Then, the strongest postcondition of ϕ with respect to the function call is obtained as $\exists x_0, y_0.\ (\phi[x_0/x, y_0/y] \wedge \varphi)$. In this example, this yields the strongest postcondition $x = 3 \wedge y = 1$.

The technique described in this paper extracts code snippets annotated with assertions and assumptions and presents these extracted fragments as queries to client analyses. Our prototype implementation for this purpose is only semi-automatic: While our implementation outputs the code snippet to be analyzed by client analyses and what assertions and assumptions to add at which program points, it currently does not invoke client analyses automatically. Instead, we invoke external analyses manually and input the results to these queries back into our analysis. While this step is, in principle, easy to automate, it requires significant engineering effort in practice because existing tools take the program in different formats and have different ways of annotating assumptions.

The technique described in this paper can utilize dynamic information by ruling out some solutions to abductive inference problems. In our prototype implementation, we manually record some values that arise in concrete executions and require the abductive solution to be consistent with these observed values.

Table 1 Experimental results on micro benchmarks

Name	LOC	Time(s)	# query	Poly.	Cong.	Blast	Compass	RP-provable?
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	✓	×	✓	✓	×

"Poly." denotes the polyhedra abstract domain, and "Cong." abbreviates the linear congruences abstract domain. The column labeled "RP-provable?" shows whether the benchmark is provable using the reduced product of the polyhedra and linear congruences abstract domains

8 Experimental evaluation

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]. Among these analyses, polyhedra analysis infers linear inequalities between integer variables, while the linear congruences abstract domain infers divisibility predicates. Unlike the analyses implemented in Interproc, BLAST is not based on abstract interpretation and uses counterexample-guided abstraction refinement. Compass focuses on heap and array reasoning and can also infer linear equalities, including disjunctive ones.

While BLAST and Compass can both analyze programs written in C, Interproc analyzes pointer-free programs written in the "Simple" language. Thus, when using Interproc to prove subgoals, we needed to rewrite the extracted fragment to comply with the syntactic restrictions of the Simple language.

The results of the first experiment are summarized in Table 1. This experiment involves 10 synthetic benchmarks available from http://www.cs.utexas.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. Thus, verifying the assertions in these examples requires a deep, and often cyclic, interaction between different tools. Using the technique proposed in this paper, all ten benchmarks can be verified using BLAST, polyhedra, linear congruences, and Compass as clients.

In Table 1, the column labeled LOC shows the number of lines of code in each benchmark, and the column labeled "Time" shows analysis time in seconds. Although the time reported here includes the time needed to compute pre- and post-conditions, performing abduction, and generating proof subgoals, it does not include the time that client analyses take to answer queries. The reason for this is that BLAST does not terminate on some of the queries we make; thus we fixed a time-out limit of 5 sec per query for client analyses. All queries made to clients with the exception of those on which BLAST does not terminate took less than half a second.

The next column labeled "# query" 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. For example, for Benchmark 1, the polyhedra abstract domain and the BLAST model checker could successfully verify at least one of the queries, while Compass and linear congruences abstract domain could not answer any.

The main point of the first experiment is that all benchmarks from Table 1 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. Table 1 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. On average, each query could be answered by 1.7 tools.

Table 2 Experimental results on open-source software

Name	LOC	Time(s)	# Query	Avg # vars per query	Avg LOC per query
Wizardpen 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

In a second experiment, summarized in Table 2, 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.

Table 2 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.

Another interesting aspect of our approach is that the compositional proofs synthesized by our technique are fairly intuitive and could potentially be useful as annotations at the source code level. In particular, since our abduction procedure always starts with a simplest solution, the lemmas presented to the client analyses are generally compact and easily understandable by humans. Hence, the proof strategies synthesized by our approach appear to be similar to manual program proofs.

9 Related work

Compositional verification The technique presented here is similar to other techniques for compositional verification such as [1,2,15]. Specifically, [1,2] 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. Generally speaking, our purpose here is different from previous work on compositional verification in that we try to combine the strengths of different verifiers in one unified framework.

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. Furthermore, as demonstrated in our experimental results, only 2 of the 10 synthetic benchmarks can be verified by the reduced product of polyhedra and linear congruences.

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 and software engineering [3,18,19,22]. The work of Gulwani et al. uses abductive inference to compute under-approximating logical operators for building universally quantified abstract domains [18]. Among these approaches, [3,19] 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.

10 Conclusion and future work

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. Our technique also helps address scalability concerns since each lemma requires proving the correctness of a small fragment of the original program. We have implemented the proposed technique, and our experiments show that it can verify programs that cannot be proven by individual tools.

In future work, we believe the techniques described in this paper can be extended in several ways: First, our PROOFSEARCH algorithm employs a simple depth-first search for synthesizing compositional proofs. While our algorithm fixes a depth limit on the search to guarantee termination, other search strategies, such as iterative deepening, might be more effective and improve the scalability of the overall approach. Another interesting avenue is to use dynamic information (or static underapproximations) to reject candidate invariants speculated using abduction. Finally, we plan to explore more efficient algorithms for performing abductive inference and extending it to richer first-order theories that do not admit quantifer elimination.

Acknowledgments We would like to thank Hongseok Yang, Aaron Bradley, Peter O'Hearn, Noam Rinetzky, and the anonymous reviewers of TACAS'13 and STTT'15 for their helpful feedback.

References

1. Cobleigh, J., Giannakopoulou, D., Păsăreanu, C.: Learning assumptions for compositional verification. TACAS, pp. 331–346 (2003)
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), 289–300 (2009)
4. Peirce, C.: Collected papers of Charles sanders peirce. Belknap Press, Cambridge (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, pp. 235–239 (2003)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, ACM, pp. 84–96 (1978)
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, pp. 169–192 (1991)
12. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. ESOP, weak updates. In (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, pp. 269–282 (1979)
15. McMillan, K.: Verification of infinite state systems by compositional model checking. Correct Hardware Design and Verification Methods, pp. 705–705 (1999)
16. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: ACM SIGPLAN Notices, ACM, vol. 41, pp 376–386 (2006)
17. Charlton, N., Huth, M.: Hector: Software model checking with cooperating analysis plugins. In: Computer Aided Verification, Springer, pp. 168–172 (2007)
18. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM, pp. 235–246 (2008)
19. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proceedings of the 1994 International Symposium on Logic programming, Citeseer, pp. 377–391 (1994)
20. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)
21. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: on-line constraint simplification for scalable static analysis. In: Static Analysis Symposium (2010)
22. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Using abduction and induction for operational requirements elaboration. In: Journal of Applied Logic (2009)