Inductive Invariant Generation via Abductive Inference

Isil Dillig
Department of Computer Science
College of William & Mary
idillig@cs.wm.edu

Thomas Dillig
Department of Computer Science
College of William & Mary
tdillig@cs.wm.edu

Boyang Li
Department of Computer Science
College of William & Mary
bli01@email.wm.edu

Ken McMillan
Microsoft Research
kenmcmil@microsoft.com

Abstract

This paper presents a new method for generating inductive loop invariants that are expressible as boolean combinations of linear integer constraints. The key idea underlying our technique is to perform a backtracking search that combines Hoare-style verification condition generation with a logical abduction procedure based on quantifier elimination to speculate candidate invariants. Starting with true, our method iteratively strengthens loop invariants until they are inductive and strong enough to verify the program. A key feature of our technique is that it is lazy: It only infers those invariants that are necessary for verifying program correctness. Furthermore, our technique can infer arbitrary boolean combinations (including disjunctions) of linear invariants. We have implemented the proposed approach in a tool called HOLA. Our experiments demonstrate that HOLA can infer interesting invariants that are beyond the reach of existing state-of-the-art invariant generation tools.

1. Introduction

The automated inference of numeric loop invariants is a fundamental program analysis problem with important applications in software verification, compiler optimizations, and program understanding. Prominent approaches for automatic generation of loop invariants include abstract-interpretation [1–4], constraint-based techniques [5, 6], counterexample guided abstraction refinement (CEGAR) [7, 8], and interpolation-based approaches [9–11].

In this paper, we present a new method for automatically generating inductive loop invariants which are expressible as boolean combinations of linear integer constraints over program variables. A loop invariant is said to be *inductive* if it is implied by the loop's precondition and is preserved in each iteration of the loop's body. Since the correctness of inductive loop invariants can be checked locally given the loop precondition, inductive invariants play a key role in many program verification systems [12–14].

Our approach to loop invariant generation is intended mainly for program verification. Therefore, similar to methods like CEGAR, our goal is to *lazily* infer only those invariants that are necessary for showing program correctness. Furthermore, we focus on numeric loop invariants in this paper because such invariants play a crucial rule in many basic verification tasks such as proving the absence of array-out-of-bounds exceptions or buffer overflows.

A salient feature of our technique is that it can naturally infer invariants that are arbitrary boolean combinations of linear integer constraints. Here, by linear integer constraints, we mean linear inequalities over integers as well as divisibility predicates. Furthermore, the invariants inferred by our method can involve disjunctions and implications. For instance, our technique is capable of inferring complex invariants that are expressible using the following formula:

$$(x\%2 = 0 \Rightarrow x + 2y < n) \wedge (x\%2 = 1 \Rightarrow x + 2y = n)$$

Another advantage of our method is that it does not require users to specify a pre-defined syntactic template over which to express the invariants. For example, some loop invariant inference techniques such as [5, 6] require the user to annotate a template describing the shape of the invariants to be inferred such $ax+by \leq c$ and then solve for the unknown

parameters a, b, c. In contrast, our technique automatically infers the shape of the required invariants without requiring hints or annotation from the user.

1.1 Overview of the Approach

The key idea underlying our approach to loop invariant inference is to perform a backtracking search that combines Hoare-style program reasoning with *logical abduction*, which is the inference of missing hypotheses for a given conclusion. Starting with the weakest possible but always correct loop invariant *true*, our technique sets up verification conditions whose validity guarantees the correctness of the program. If a verification condition (VC) is invalid, our technique employs abductive inference to fix this invalid VC, which corresponds to the speculation of a new loop invariant. The current speculation is used to strengthen the existing invariants, and new VCs are set up to check the correctness of the speculation. This process continues until either all the VCs become valid or we derive a contradiction. The former outcome means that we have found inductive loop invariants strong enough to show the correctness of the program and our algorithm terminates. The latter outcome means that we have made a wrong speculation; therefore, we backtrack and try a different inference.

In our approach, the generated verification conditions are conjunctions of clauses of the form $\chi \Rightarrow \gamma$, asserting the inductiveness of loop invariants and correctness of assertions in the program. Specifically, the right hand side γ of the implication is a goal we want to show, such as a loop post-condition. On the other hand, the left-hand-side χ encodes known facts about the program, such as already inferred loop invariants.

Now, given an invalid clause $\chi \Rightarrow \gamma$ of the VC, our technique tries to fix this "broken" clause by strengthening the left-hand side of the implication. In particular, we want to infer a strengthening ψ of the current loop invariant χ such that:

1. $\models (\chi \wedge \psi) \Rightarrow \gamma$
2. $\text{SAT}(\chi \wedge \psi)$

Here, condition (1) states that the strengthening ψ is sufficient to make a particular clause of the VC valid. Condition (2) states that the strengthening ψ must be consistent with the left-hand side χ, since χ encodes known (or speculated) invariants about the program. The inference of a formula ψ satisfying these two requirements is an instance of *logical abduction*, as defined by Peirce [15]. Therefore, a main insight underlying our approach is that logical abduction is useful for finding suitable strengthenings of loop invariants that are sufficient for making the program's verification condition valid.

Now, while our inferred strengthening ψ fixes an invalid clause of the VC, it is nonetheless a speculation and may or may not correspond to a correct loop invariant. Therefore, to ensure the soundness of our approach, our technique generates new VCs at every step encoding the correctness of all invariants, including ψ. Therefore, if ψ is an incorrect speculation, the new VC will contain an invalid clause that needs to be fixed. If the algorithm reaches a point where a given VC clause can no longer be fixed through abductive strengthening (for instance, when $\chi \Rightarrow \gamma$ is unsatisfiable), this means we have made a wrong speculation and we must backtrack. On the other hand, if all VC clauses become valid, this means all of our speculations are correct. Thus, when the algorithm terminates, we have identified correct inductive loop invariants that are strong enough to show the correctness of the program. However, our technique does not have termination or completeness guarantees; it is indeed possible for the algorithm to diverge in an infinite chain of speculations.

In principle, this combination of Hoare-style VC generation and logical abduction can be used to infer any class of invariants. However, since our algorithm for performing abduction is based on quantifier elimination, this paper focuses only on the inference of loop invariants expressible in Presburger arithmetic.

1.2 Organization and Contributions

The rest of this paper is organized as follows: Section 2 walks the reader through a small example illustrating how our technique infers loops invariants. Section 3 presents a small imperative language that we use for the formal development, and Section 4 presents the main invariant inference algorithm. Section 5 discusses extensions to the basic algorithm and describes our implementation. Section 6 presents an experimental evaluation of our approach, comparing our tool HOLA with three other loop invariant generation tools. Finally, Sections 7 and 8 discuss related work and highlight future research directions.

To summarize, this paper makes the following key contributions:

- We present a novel algorithm based on backtracking search for automatically inferring inductive loop invariants that are expressible as boolean combinations of linear integer constraints.
- We show how Hoare-style program reasoning and a logical abduction procedure based on quantifier elimination can be combined for linear invariant generation. To the best of our knowledge, this is the first application of logical abduction for automatic generation of numeric loop invariants.
- We have implemented the proposed technique in a tool called HOLA and present a comparison between HOLA and other available state-of-the-art invariant generation tools, namely BLAST, InvGen, and Interproc. Our experiments on a set of loop invariant generation benchmarks show that our approach is promising for advancing the state-of-the-art in numeric loop invariant generation.

```
void foo(int flag) {
1.    int i, j, a, b;
2.    a = 0; b = 0; j = 1;
3.    if(flag) i=0; else i=1;
4.    while(*) [φ] {
5.          a++; b+=(j-i); i+=2;
6.          if(i%2 == 0) j+=2; else j++;
7.    }
8.    if(flag) assert(a==b);
}
```

Figure 1. Code example to illustrate our approach

2. Example Illustrating our Technique

We illustrate the main ideas of our technique using the code example shown in Figure 1. Here, the annotation $[\phi]$ next to the while loop at line 4 means that ϕ is a placeholder variable representing an unknown invariant of this loop. Our goal in this section is to infer a concrete formula φ for the placeholder ϕ such that φ is both inductive as well as strong enough to prove the assertion at line 8.

Our technique starts by instantiating the placeholder variable ϕ with the weakest possible but always correct loop invariant, namely *true*. Using Hoare-style reasoning, we then generate verification conditions for the program, which, in this case, yields:

$$\text{true} \Rightarrow (\text{flag} \Rightarrow a = b)$$

This VC stipulates that the current loop invariant *true* is strong enough to imply the loop post-condition, which is $\text{flag} \Rightarrow a = b$. Clearly, this VC is not valid, therefore, we attempt to fix this broken VC by finding an abductive strengthening ϕ of the left-hand side such that:

$$(\text{true} \wedge \phi) \Rightarrow (\text{flag} \Rightarrow a = b)$$

There are many solutions to this abduction problem, including the solutions $\neg$flag, $a = b$, and flag $\Rightarrow a = b$ computed by our abduction algorithm. To keep our discussion concise, let us start with the solution $\phi = (\text{flag} \Rightarrow a = b)$, which means $(\text{flag} \Rightarrow a = b)$ is our speculated loop invariant.

We now proceed by generating new VCs that assert the correctness of our candidate invariant. This results in the generation of the following new VC:

$$(\text{flag} \Rightarrow a = b) \Rightarrow (\text{flag} \Rightarrow (a + 1 = b + j - i))$$

This VC simply stipulates that our candidate invariant is inductive, as the right-hand side of the implication is the weakest precondition of the candidate invariant. Unfortunately, however, the VC is again not valid; thus, we proceed to set up another abduction problem to find a suitable strengthening ϕ of the loop invariant:

$$(\text{flag} \Rightarrow a = b \ \wedge \ \phi) \Rightarrow (\text{flag} \Rightarrow (a + 1 = b + j - i)) \ \ (*)$$

Our algorithm for performing abduction again computes several solutions, including $\neg$flag, $j = i + 1$, and flag $\Rightarrow$ $j = i + 1$.

Now, suppose we initially pick a wrong solution, say $j = i + 1$. This means our new candidate loop invariant is now $j = i+1 \wedge (\text{flag} \Rightarrow a = b)$. Since $j = i+1$ does not hold initially (i.e., the weakest precondition of $j = i+1 \wedge (\text{flag} \Rightarrow a = b)$ with respect to lines 1-3 is invalid), we therefore reject this candidate and backtrack to our speculation from the previous level, namely flag $\Rightarrow a = b$.

Now, having backtracked to the previous decision level, we try another solution to our abduction problem from (*). Suppose that we now choose the solution:

$$\phi = (\text{flag} \Rightarrow j = i + 1)$$

Combining this strengthening with the existing invariant flag $\Rightarrow a = b$, we now obtain the following candidate loop invariant:

$$\mathcal{I} \ = \ (\text{flag} \Rightarrow (a = b \wedge j = i + 1))$$

It turns out that this new speculation $\mathcal{I}$ is now correct, but, unfortunately, $\mathcal{I}$ is still not inductive because the following VC clause asserting the inductiveness of $\mathcal{I}$ is not valid:

$$\begin{array}{c}(\text{flag} \Rightarrow (a = b \wedge j = i + 1)) \\ \Rightarrow \\ (\text{flag} \Rightarrow (a + 1 = b + j - i \wedge (i\%2 = 0 \Rightarrow j = i + 1) \wedge \\ (i\%2 \neq 0 \Rightarrow i = j)))\end{array}$$

Here, note that the right-hand side of the outer-level implication corresponds to the weakest precondition of $\mathcal{I}$. To make this VC valid, we now solve a final abduction problem to find yet another strengthening of $\mathcal{I}$. In this case, one of the solutions we obtain is flag $\Rightarrow i\%2 = 0$. Thus, we obtain a new candidate loop invariant $\mathcal{I}'$ by conjoining this solution with $\mathcal{I}$:

$$\mathcal{I}' \ = \ (\text{flag} \Rightarrow (a = b \wedge j = i + 1 \wedge i\%2 = 0))$$

At this point, the new VC becomes valid. In other words, we have shown that $\mathcal{I}'$ is a correct inductive invariant strong enough to prove the assertion at line (8).

3. Language

In this section, we give a simple imperative language that we use for formalizing our technique:

Program π	:=	s
Statement s	:=	skip | $v := e$
		| $s_1; s_2$ | choose s_1 s_2
		| while C $[\phi]$ do $\{s\}$
		| assert p | assume p
Expression e	:=	v | int | $e_1 + e_2$
		| $e *$ int | $e\%$int
Conditional C	:=	$e_1 \odot e_2$ $(\odot \in \{<, >, =\})$
		| $C_1 \wedge C_2$ | $C_1 \vee C_2$ | $\neg C$

```
procedure INVGEN(π):
    input: program π
    output: mapping Δ from each placeholder φ_i
            to a concrete loop invariant ψ_i

(1)  let Δ = [φ_i ↦ true | φ_i ∈ invs(π)]
(2)  Δ' = VERIFY(π, Δ)
(3)  return Δ'

procedure VERIFY(π, Δ):
    input: program π and mapping Δ
    output: new mapping Δ' from each placeholder φ_i
            to a concrete loop invariant ψ_i

(4)  (χ, φ) = VCGEN(π, Δ)
(5)  if ⊭ χ return ∅
(6)  if ⊨ elim(φ) return Δ
(7)  let φ_i ∈ clauses(φ) such that ⊭ elim(φ_i)
(8)  (φ, S) = ABDUCE(φ_i)
(9)  for each ψ_i ∈ S
(10)     ψ = Δ(φ) ∧ ψ_i
(11)     Δ' = VERIFY(π, Δ[φ ↦ ψ])
(12)     if Δ' ≠ ∅ return Δ'
(13) done
(14) return ∅
```

Figure 2. The main invariant generation algorithm

In this language, programs consist of one or more statements. Statements include skip, assignments, sequencing, choose statements (which non-deterministically execute s_1 or s_2), while loops, assertions, and assumptions. Expressions include variables v, integer constants int, addition $(e_1 + e_2)$, linear multiplication ($e*$int), and mod expressions $e\%$int. Finally, conditionals can be comparisons between expressions as well as logical conjunction, disjunction, and negation.

Loops in this language are decorated with unique placeholder formulas ϕ which represent unknown loop invariants but have no effect on the program semantics. For each placeholder ϕ, the goal of our technique is to infer a concrete logical formula ψ such that ψ is both inductive as well as strong enough to imply the loop postcondition. In the remainder of the paper, we use the notation $\text{invs}(\pi)$ to refer to all placeholders ϕ used in π.

4. Algorithm for Generating Inductive Loop Invariants

Our algorithm INVGEN for generating inductive loop invariants is shown in Figure 2. It takes as input a program π that we want to verify and outputs a mapping Δ from each placeholder invariant ϕ in π to an inductive loop invariant ψ. If we cannot verify the program, then the algorithm returns the empty mapping $\emptyset$.

As mentioned in Section 1, our algorithm starts by initializing each ϕ to true and iteratively strengthens loop invariants until they become inductive and strong enough to verify the program. Thus, at line 1, we initialize Δ by mapping all placeholders ϕ_i to true. The iterative strengthening of loop invariants in Δ is performed by the recursive VERIFY procedure, also given in Figure 2.

In the VERIFY procedure, the input Δ represents our current set of speculated loop invariants. Using these candidate invariants Δ, we then invoke a VCGEN procedure to compute the weakest precondition χ of program π as well as its VC φ (line 4). We defer discussion of the VCGEN procedure to Section 4.1. The generated VC φ asserts that each candidate invariant ψ in Δ is inductive and that it implies the loop postcondition. Specifically, the generated VC's are conjunctions of clauses of the form $(\beta \wedge \phi) \Rightarrow \gamma$. Here, ϕ is a placeholder for a potential strengthening of the current loop invariant. Formulas β and γ do not contain any placeholders and are generated using the candidate invariants given by Δ. If the VC φ with all placeholders ϕ replaced by true is valid, this means the current Δ is a solution to our verification problem.

Line 5 of VERIFY checks the validity of the program's weakest precondition χ. If χ is not valid, this means our candidate invariants given by Δ are not correct. In this case, we return $\emptyset$ to indicate failure, which causes backtracking in the overall algorithm. Next, at line 6, we check whether the current VC φ is valid. As mentioned before, φ contains placeholder variables ϕ, which represent potential strengthenings of the current loop invariants. Therefore, to check the validity of the VC, we need to replace all placeholders ϕ with true. For this purpose, we use the notation $\text{elim}(\varphi)$ to indicate the substitution of ϕ with true in φ. If $\text{elim}(\varphi)$ is valid, we have found a set Δ of inductive loop invariants strong enough to verify the program, and we therefore return Δ as our solution.

On the other hand, if $\text{elim}(\varphi)$ is not valid, this means our current loop invariants are not strong enough, and we need to strengthen them further. For this purpose, we select one invalid clause φ_i of the VC containing one placeholder variable ϕ and attempt to fix it. Specifically, recall that φ_i is of the form $(\beta \wedge \phi) \Rightarrow \gamma$ where β and γ do not contain any placeholders. To fix clause φ_i, we call the procedure ABDUCE to find a set S of suitable strengthenings for the left-hand side. As we will see in Section 4.2, each $\psi_i \in S$ is guaranteed to make φ_i valid, but ψ_i may or may not be a valid invariant. Therefore, at line 11, we recursively invoke VERIFY to check the correctness of the speculated invariant. In the recursive invocation, we add our current speculation ψ_i to Δ. Specifically, since ψ_i is a strengthening for the current loop invariant $\Delta(\phi)$, we conjoin ψ_i with the existing $\Delta(\phi)$ when making the recursive call to VERIFY. If VERIFY does not return $\emptyset$, this means we have found a suitable set of inductive loop invariants and return Δ' as the solution.

$$(1)\frac{}{\Delta, \chi \vdash \text{skip} : \chi, \text{true}} \qquad (2)\frac{}{\Delta, \chi \vdash v := e : \chi[e/v], \text{true}}$$

$$(3)\frac{}{\Delta, \chi \vdash \text{ assert } C : \chi \wedge C, \text{true}} \qquad (4)\frac{}{\Delta, \chi \vdash \text{assume } C : C \Rightarrow \chi, \text{true}}$$

$$(5)\frac{\begin{array}{c}\Delta, \chi \ \vdash s_2 : \chi_2, \varphi_2 \\ \Delta, \chi_2 \vdash s_1 : \chi_1, \varphi_1\end{array}}{\Delta, \chi \vdash s_1; s_2 : \chi_1, \varphi_1 \wedge \varphi_2} \qquad (6)\frac{\begin{array}{c}\Delta, \chi \vdash s_1 : \chi_1, \varphi_1 \\ \Delta, \chi \vdash s_2 : \chi_2, \varphi_2\end{array}}{\Delta, \chi \vdash \text{choose } s_1 \ s_2 : \chi_1 \wedge \chi_2, \varphi_1 \wedge \varphi_2}$$

$$(7)\frac{\begin{array}{c}\Delta, \Delta(\phi) \vdash s : \chi', \varphi' \\ \varphi_1 \ = \ (\neg C \wedge \Delta(\phi) \wedge \phi) \Rightarrow \chi \\ \varphi_2 \ = \ (C \wedge \Delta(\phi) \wedge \phi) \Rightarrow \chi'\end{array}}{\Delta, \chi \vdash \text{while } C \ [\phi] \text{ do } \{s\} : \Delta(\phi), \varphi_1 \wedge \varphi_2 \wedge \varphi'}$$

Figure 3. VC Generation

Of course, it is also possible that our current strengthening ψ_i does not lead to a valid proof; this is indicated by the recursive invocation of VERIFY returning $\emptyset$. In this case, we backtrack from our current speculation and try the next solution to the abduction problem defined by $(\chi \wedge \phi) \Rightarrow \gamma$. If we exhaust all abductive solutions without finding a valid proof, our current proof attempt has failed, and we therefore return $\emptyset$ to indicate failure.

4.1 VC Generation

Our VC generation procedure VCGEN is presented as a set of inference rules in Figure 3. The rules produce judgments of the form:

$$\Delta, \chi \vdash s : \chi', \varphi$$

Here, Δ is a candidate invariant represented as a mapping from each placeholder ϕ to a formula ψ. The VC φ is a conjunction of clauses of the form $(\beta \wedge \phi) \Rightarrow \gamma$ where β, γ are placeholder-free formulas and ϕ is a placeholder representing a possible strengthening of the current loop invariant. The meaning of this judgment is that, if $\text{elim}(\varphi)$ (the VC with all placeholders replaced by *true*) is valid, then $\{\chi'\}\ s\ \{\chi\}$ is a valid Hoare triple.

Rules (1)-(6) in Figure 3 describe a standard weakest precondition computation; hence we omit the explanation for these rules. Since Rule (7) for while loops is non-standard, we only elaborate on this rule.

For generating the VC and the weakest precondition of the while loop, Rule (7) utilizes the current speculated invariants given by Δ. Specifically, the weakest precondition for the while loop is $\Delta(\phi)$, which represents the current speculated invariant for the loop. In the rule, χ' is the weakest precondition for the loop body s to establish the candidate loop invariant $\Delta(\phi)$. Clause φ_1 of the VC says that on loop exit, the candidate invariant $\Delta(\phi)$, strengthened with ϕ, implies the loop's post-condition χ. Clause φ_2 of the VC says that executing the loop when ϕ holds re-establishes the invariant $\Delta(\phi)$. In particular, if ϕ is true, these are just the standard VC's for a while loop. The addition of the placeholder ϕ allows us to weaken the VC by adding an assumption, which we obtain by abduction.

THEOREM 1 (Soundness). *If $\Delta, \chi \vdash s : \chi', \varphi$ is derivable and $\text{elim}(\varphi)$ is valid, then $\{\chi\}\ s\ \{\chi'\}$ is valid.*

PROOF 1. *The proof for rules (1)-(6) follow immediately from standard Hoare logic; we only give the proof for rule (7). Assuming that $\text{elim}(\varphi_1 \wedge \varphi_2 \wedge \varphi')$ is valid we need to show:*

$$\{\Delta(\phi)\}\ \textit{while } C\ [\phi]\ \textit{do}\ \{s\}\ \{\chi\} \qquad (1)$$

By inductive hypothesis, we have $\{\chi'\}\ s\ \{\Delta(\phi)\}$. Since $\text{elim}(\varphi_2)$ implies $C \wedge \Delta(\phi) \Rightarrow \chi'$, we have $\{C \wedge \Delta(\phi)\}\ s\ \{\Delta(\phi)\}$ by precondition strengthening. Using the standard Hoare logic rule for while loops, we obtain:

$$\{\Delta(\phi)\}\ \textit{while } C\ \textit{do}\ \{s\}\ \{\Delta(\phi) \wedge \neg C\} \qquad (2)$$

Since $\text{elim}(\varphi_1)$ is valid, we have $\neg C \wedge \Delta(\phi) \Rightarrow \chi$. Finally, by applying postcondition weakening to (2), we obtain (1).

4.2 Generating Candidate Strengthenings via Abduction

In this section, we describe an ABDUCE procedure for finding possible strengthenings for the current candidate loop invariant. The procedure ABDUCE takes as input a VC clause of the form $(\chi \wedge \phi) \Rightarrow \gamma$ where ϕ is a placeholder and infers a strengthening ψ to plug in for ϕ such that:

1. $\models (\chi \wedge \psi) \Rightarrow \gamma$
2. $\text{SAT}(\chi \wedge \psi)$

One obvious -but not particularly useful- solution to this abduction problem is $\phi = \gamma$. To see why γ is not a useful solution, consider the VC clause that asserts the inductiveness of some candidate invariant I for a loop with body s:

$$I \wedge C \Rightarrow wp(s, I)$$

Hence, in our setting, the abductive solution γ corresponds to strengthening the current loop invariant I with its weakest precondition $wp(s, I)$. However, in general, $wp(s, I)$ is too weak of a strengthening and typically leads to divergence. In fact, starting with the loop postcondition I and repeatedly conjoining it with $wp(s, I)$ is equivalent to unrolling the loop body!

Therefore, we are interested in solutions to the abduction problem that are logically stronger than just the weakest precondition of the current invariant. The main strengthening mechanism we will use here is quantifier elimination. To see how we can use quantifier elimination to solve our abduction problem, we first observe that the entailment

$$\models (\chi \wedge \psi) \Rightarrow \gamma$$

can be rewritten as:

$$\psi \models \chi \Rightarrow \gamma$$

Now, consider any subset of the free variables V of the formula $\chi \Rightarrow \gamma$. Clearly, we have:

$$\forall V.\ \chi \Rightarrow \gamma \models \chi \Rightarrow \gamma$$

Therefore, any formula φ that is logically equivalent to $\forall V.\chi \Rightarrow \gamma$ and that does not contradict χ is a solution to our abduction problem.

DEFINITION 1. *(Universal subset) We call a set of variables V a* universal subset *(US) of ϕ with respect to ψ if the formula $(\forall V.\phi) \wedge \psi$ is satisfiable.*

Hence, if V is a universal subset of $\chi \Rightarrow \phi$ with respect to χ, we can obtain a solution to our abduction problem by projecting out variables V from the formula $\chi \Rightarrow \phi$ through universal quantifier elimination in Presburger arithmetic. Furthermore, empirically, it turns out that solutions obtained in this way through quantifier elimination are very useful candidates for auxiliary loop invariants. Intuitively, quantifier elimination is useful because it allows us to project out variables that are irrelevant and prevent the invariant from being inductive.

EXAMPLE 1. *To get some intuition about why quantifier elimination is useful for generating inductive invariants, consider the following code example:*

```
int x = 0; int y = 0;
while(x < n) {
      x = x+1;
      y = y+2;
}
assert(y >= n);
```

Assuming the initial loop invariant is true, the generated VC is $x \geq n \Rightarrow y \geq n$, which does not correspond to an inductive invariant. Now, if we project out n from this formula by universally quantifying n and applying quantifier elimination, we obtain $y \geq x$, which is indeed an inductive loop invariant. Intuitively, here, quantifier elimination allows us to generalize the last iteration of the loop to an inductive assertion.

procedure ABDUCE(φ):
input: formula φ of the form $\chi \wedge \phi \Rightarrow \gamma$
output: (ϕ, S) such that $\models \chi \wedge \psi_i \Rightarrow \gamma$ for every $\psi_i \in S$

(1) let $\phi = (\chi \wedge \phi) \Rightarrow \gamma$
(2) let $\theta = \{\chi\}$
(3) let $S = [\,]$
(4) while true
(5) $\quad V = \text{MUS}(\chi \Rightarrow \gamma, \theta)$
(6) $\quad$ if $V = \emptyset$ then break
(7) $\quad \psi = \text{QE}(\forall V.\chi \Rightarrow \gamma)$
(8) $\quad S = S :: \psi$
(9) $\quad \theta = \theta \cup \{\neg\psi\}$
(10) done
(11) return (ϕ, S)

Figure 4. Algorithm for computing an ordered list S of solutions to abduction problem $\chi \wedge \phi \Rightarrow \gamma$. The procedure $\text{MUS}(\phi, \chi)$ computes a maximum universal subset of ϕ with respect to χ.

Based on this observation, Figure 4 summarizes the full abduction algorithm that we use for generating candidate strengthenings of the current loop invariant. Effectively, this algorithm generates all possible universal subsets of $\chi \Rightarrow \gamma$ with respect to χ and infers a candidate strengthening by projecting out all variables in that universal subset from our verification condition.

An important consideration in this algorithm is which universal subset to consider first. Specifically, since the invariant generation algorithm from Figure 2 performs a depth-first search, candidate strengthenings that are too weak can cause our algorithm to either take too long or, worse, diverge in an infinite chain of speculations. On the other hand, if we first try a strengthening that is too strong, this is often not a problem because the algorithm can quickly derive a contradiction and backtrack.

Now, observe that for a pair of universal subsets U, U' of some formula ϕ, if $U \supset U'$, then $\forall U.\phi$ logically implies $\forall U'.\phi$. Hence, to generate stronger auxiliary invariants before weaker ones, our abduction algorithm considers universal subsets in decreasing order of their cardinality.

DEFINITION 2. *(Maximum universal subset) A universal subset U of ϕ is a maximum universal subset (MUS) if $|U| \geq |U'|$ for any other universal subset U' of ϕ.*

An algorithm for computing maximum universal subsets of formulas is described in our earlier work [16].

We are now ready to explain the full algorithm shown in Figure 4 for generating an ordered list S of candidate strengthenings. In each iteration of the while loop, we compute a maximum universal subset of $\chi \Rightarrow \gamma$ consistent with all constraints in set θ, which initially only includes χ. If there does not exist an MUS of $\chi \Rightarrow \gamma$ consistent with all constraints in θ, our algorithm terminates and returns S as a list of candidate strengthenings. However, if there exists an MUS V, we then obtain a new candidate invariant ψ by projecting out all variables in V from the formula $\chi \Rightarrow \gamma$ using a quantifier elimination procedure for Presburger arithmetic, such as Cooper's method [17]. Now, since we want to obtain a different MUS in the next iteration, we add $\neg\psi$ to θ. This strategy ensures that the MUS obtained in the next iteration is different from previous ones because V can no longer be an MUS consistent with $\neg\psi \equiv \neg(\forall V.(\chi \Rightarrow \gamma))$. Therefore, the list S computed by our ABDUCE procedure contains the set of auxiliary candidate invariants in decreasing order of logical strength.

Observe that the abduction algorithm we present here is not complete. That is, for an abduction problem defined by χ, γ, our method does not generate all possible formulas ψ such that $\chi \wedge \psi \Rightarrow \gamma$ is valid. This is not surprising since there may be infinitely many ψ that solve a given abduction problem in Presburger arithmetic. Instead, our approach only computes those solutions that can be obtained through applying quantifier elimination on the original formula. In our experimental evaluation (see Section 6), there are some benchmarks where our technique is unsuccessful due to this source of incompleteness.

5. Improvements and Extensions

We have implemented the algorithm described in this paper in a tool called HOLA [1]. HOLA currently works on the pointer-free fragment of the C language and uses the SAIL front-end for converting C programs to an intermediate language [18]. HOLA also utilizes the Mistral SMT solver [16, 19, 20] for determining satisfiability, performing quantifier elimination in Presburger arithmetic, and computing maximum universal subsets.

An important difference between our implementation and the basic algorithm described in this paper is that our implementation performs a dual forwards and backwards analysis rather than pure backwards reasoning. Specifically, while the algorithm described in this paper computes only weakest preconditions, our implementation simultaneously computes strongest postconditions and weakest preconditions. This combination of forwards and backwards reasoning has two significant advantages. First, the computation of strongest postconditions allows us to obtain (overapproximate) loop preconditions which are then used to reject abductive solutions that violate facts known to hold on loop entry. For example, if we know that $x = 0$ and $y = 1$ hold on loop entry, we can immediately reject the abductive speculation $x = y$ since it contradicts the loop precondition. Hence, the use of forward reasoning allows us to locally reject many wrong speculations and prevents unnecessary backtracking.

A second important advantage of performing forwards reasoning is that it allows us to obtain stronger initial loop invariants than just the trivial invariant *true*. Specifically, recall that the INVGEN algorithm from Figure 2 initializes Δ to map each ϕ_i to true. Now, if we know that the loop precondition is P and that variables V are not modified in the loop, then we can safely initialize the loop invariant to $\exists\overline{V}.\ P$ where $\overline{V}$ represents variables that may be modified in the loop. As an example, suppose that the precondition for some loop is $x + y \geq n \wedge y \leq 0$. If we know that y may be modified in the loop, but x is definitely not modified, then we can safely conclude that $x \geq n$ is an invariant of the loop by applying existential quantifier elimination to the formula $\exists y.(x + y \geq n \wedge y \leq 0)$. This strategy gives us a sound initial invariant that is better than just true and therefore cuts down substantially on the number of search paths explored by our algorithm.

Another improvement of our implementation over the basic algorithm is the lazy recomputation of verification conditions. Specifically, recall that the VERIFY algorithm from Figure 2 recomputes the VC for the entire program in each recursive invocation, which is both expensive and unnecessary. In our implementation, we therefore only recompute those clauses of the VC that could have changed due to strengthening a given loop invariant. For example, if we strengthen the invariant of some loop L, this only changes the VC of L, loops that immediately come before and after L, loops in which L is nested, and loops that are nested inside L. Based on this observation, our implementation updates VCs locally rather than recomputing the VC for the entire program.

Our implementation also differs from the presentation in this paper in that it does not eagerly generate all possible solutions to an abduction problem (as was done in the ABDUCE algorithm from Figure 4). Specifically, our implementation generates one abductive solution at a time and immediately tries to verify this speculation. A new abductive solution is generated lazily only when the previous speculation is wrong and forces backtracking to the previous decision level.

6. Experimental Evaluation

To evaluate the proposed technique, we compared our tool HOLA with three available state-of-the-art invariant generation tools, namely BLAST [7], InvGen [5], and Interproc [21]. Each of the three tools we compared against represents a different family of invariant generation techniques: Interproc is based on abstract interpretation and, in our experiments, we compared against the most expressive abstract domain implemented by Interproc, which is the re-

[1] HOLA stands for "HOare Logic with Abduction"

Name	LOC	BLAST	Time(s)	InvGen	Time(s)	Interproc	Time(s)	HOLA	Time(s)
Benchmark 1	21	✔	0.10	✔	0.14	✔	0.01	✔	0.03
Benchmark 2	26	✘	0.08	✘	0.05	✘	0.01	✔	0.32
Benchmark 3	22	✔	0.46	✔	0.20	✘	0.01	✔	0.04
Benchmark 4	21	✔	1.22	✘	0.07	✘	0.01	✔	0.03
Benchmark 5	27	✘	—	✔	0.21	✔	0.01	✔	0.06
Benchmark 6	28	✘	0.08	✘	0.06	✘	0.01	✔	0.25
Benchmark 7	27	✘	3.09	✔	0.22	✔	0.01	✔	0.56
Benchmark 8	30	✘	5.68	✔	0.23	✘	0.01	✔	0.59
Benchmark 9	49	✘	—	✔	0.34	✔	0.01	✔	0.06
Benchmark 10	30	✘	0.08	✘	0.06	✘	0.01	✔	0.18
Benchmark 11	24	✘	—	✔	0.16	✔	0.01	✔	0.20
Benchmark 12	34	✘	7.94	✘	—	✘	0.01	✔	3.52
Benchmark 13	25	✔	0.38	✘	0.20	✔	0.01	✔	0.38
Benchmark 14	26	✔	1.14	✔	0.15	✔	0.01	✔	1.31
Benchmark 15	28	✔	0.30	✔	0.15	✔	0.01	✘	—
Benchmark 16	23	✘	1.13	✔	0.13	✔	0.01	✔	0.12
Benchmark 17	22	✔	0.32	✔	0.19	✔	0.01	✔	0.05
Benchmark 18	23	✘	—	✘	12.05	✘	0.01	✔	3.64
Benchmark 19	24	✔	1.06	✘	0.08	✘	0.01	✘	—
Benchmark 20	33	✔	2.11	✘	0.20	✘	0.01	✔	2.29
Benchmark 21	39	✘	0.93	✘	0.04	✔	0.01	✔	1.55
Benchmark 22	26	✘	0.10	✘	0.07	✘	0.01	✔	0.23
Benchmark 23	20	✘	—	✔	0.16	✘	0.01	✔	0.03
Benchmark 24	18	✔	0.08	✔	0.22	✔	0.01	✔	0.07
Benchmark 25	33	✔	0.51	✔	4.63	✘	0.01	✔	0.07
Benchmark 26	24	✘	0.07	✘	0.07	✘	0.01	✔	0.08
Benchmark 27	23	✔	0.14	✔	0.21	✔	0.01	✔	0.08
Benchmark 28	25	✘	—	✔	0.17	✔	0.01	✔	0.05
Benchmark 29	32	✘	0.08	✘	0.05	✘	0.01	✔	0.37
Benchmark 30	22	✘	0.55	✔	0.12	✘	0.01	✔	0.03
Benchmark 31	29	✔	0.13	✔	0.25	✘	0.01	✔	0.25
Benchmark 32	24	✔	0.11	✘	0.07	✘	0.01	✔	0.64
Benchmark 33	36	✔	0.50	✘	—	✘	0.01	✔	0.10
Benchmark 34	23	✘	1.12	✘	0.05	✘	0.01	✘	—
Benchmark 35	17	✔	0.15	✘	0.10	✘	0.01	✔	0.09
Benchmark 36	71	✘	1.01	✘	0.09	✘	0.01	✔	1.00
Benchmark 37	21	✔	0.62	✘	0.14	✘	0.01	✔	0.87
Benchmark 38	20	✘	0.14	✘	0.05	✘	0.01	✔	0.32
Benchmark 39	62	✔	0.27	✔	0.28	✔	0.01	✔	0.40
Benchmark 40	30	✘	0.86	✘	0.06	✘	0.01	✔	0.94
Benchmark 41	25	✘	2.69	✔	0.16	✘	0.01	✔	0.53
Benchmark 42	37	✘	0.08	✔	0.07	✔	0.01	✔	0.07
Benchmark 43	27	✔	0.08	✔	0.18	✔	0.01	✔	0.05
Benchmark 44	35	✔	0.36	✘	0.22	✘	0.01	✔	1.25
Benchmark 45	44	✘	0.30	✘	0.11	✘	0.01	✔	0.65
Benchmark 46	24	✘	0.12	✘	0.05	✘	0.01	✔	0.18

Figure 5. Experimental results

duced product of polyhedra and linear congruences abstract domains. InvGen is a constraint-based invariant generator and solves for the unknown parameters of a given template of invariants. In our experiments, we used the default templates provided by InvGen. Finally, BLAST is a CEGAR-based model checker which uses Craig interpolation to generate candidate invariants from counterexamples.

In our experimental evaluation, we used 46 loop invariant benchmarks, each containing at least one loop and at least one assertion. Some benchmarks contain nested loops or multiple sequential loops. 26 of our 46 benchmarks are either taken directly or adapted from other sources, including examples from other loop invariant generation papers [2, 22–31], the InvGen test suite [32], and the NECLA verification benchmarks [33]. The remaining 20 benchmarks are taken from the HOLA test suite. All benchmarks are available from http://www.cs.wm.edu/~tdillig/oopsla13-benchmarks.tar.gz.

In the experiments, our goal is to determine which of the tools can infer strong enough invariants sufficient to verify all assertions in the program. While BLAST, InvGen, and HOLA can directly process assertions in the source code, Interproc only outputs inferred invariants for each program point. Thus, for Interproc, we used an SMT solver to check whether the inferred invariants imply the safety of the assertion.

Figure 5 presents the results of our experimental evaluation. These experiments were performed on an Intel i5 2.6 GHz CPU with 8 GB of memory. For each tool, ✔ indicates that the tool was able to verify all assertions in the program, while ✘ denotes the opposite (i.e., either the tool did not terminate or was unable to infer the necessary invariants). The columns labeled "Time" next to the tool names indicate how long the tools took on each benchmark in seconds. A dash (—) in this column indicates that the tool did not terminate in 200 seconds.

As Figure 5 shows, the proposed technique is quite effective at finding strong enough invariants sufficient to verify these benchmarks. Specifically, HOLA can verify 43 out of our 46 benchmarks and diverges on three. As Figure 5 also indicates, HOLA can verify 13 benchmarks that cannot be proven by any other tool. In contrast, there are two benchmarks that can be verified by at least one other tool, but not by HOLA. Overall, HOLA has a success rate of 93.5% on these benchmarks while BLAST, InvGen, and Interproc have a success rate of 43.5%, 47.8%, and 37.0% respectively.[2]

While HOLA performs overall better than the other tools on these benchmarks, the set of invariants that can be inferred by HOLA is not a superset of those invariants that can be inferred by other approaches. Indeed, HOLA fails to verify benchmark 19 which can be verified by BLAST as well as benchmark 15 which can be verified by all other tools. For these benchmarks, HOLA fails to find an inductive invariant because the required invariant can simply not be obtained by performing quantifier elimination on the generated VCs. In principle, however, a different algorithm for performing abduction could generate the required invariant. In contrast, HOLA can verify 13 benchmarks that no other tool can verify. Some of these benchmarks require disjunctive invariants and are therefore fundamentally beyond the capabilities of standard abstract interpretation tools such as Interproc. [3] InvGen fails to verify these benchmarks because the required invariants are simply not in the vocabulary of the templates used by InvGen. In contrast, BLAST diverges on many of these benchmarks since the interpolants computed from counterexample traces do not generalize into inductive invariants. We believe these results demonstrate that our new method complements existing techniques and advances the state-of-the-art in loop invariant generation.

Figure 6 presents more details about the behavior of our algorithm on the experimental benchmarks. The column labeled "'Strengthenings" shows the minimum number of strengthening steps required for computing the final invariant. In contrast, the column labeled "Iterations" shows the number of recursive calls made by our algorithm. Observe that if our search strategy is perfect (i.e., the first abductive solution always corresponds to a valid program invariant), then the number of iterations should be exactly one greater than the number of strengthenings. Therefore, a high ratio of strengthenings to iterations means that our algorithm immediately homes in on the correct invariants. The column labeled "Backtracks" reports the total number of backtracking steps performed by our algorithm. If the number of backtracking steps is high, this means we perform many unnecessary strengthenings.

According to the data from Figure 6, our algorithm takes an average of 22.4 iterations and 19.7 backtracking steps to compute the final invariants required for verifying the program. For most benchmarks, the ratio of strengthenings to iterations is high, indicating that the algorithm quickly homes in on the correct invariant. However, for some benchmarks, such as 18, 20, and 37, the strengthening to iteration ratio is low and the number of backtracking steps is high. This indicates that our simple depth-first search strategy is sometimes ineffective. That is, our algorithm always chooses strengthening the current speculated invariant over exploring other speculations, but for some benchmarks, this simple strategy results in exploring many dead ends. This data indicates that the performance of our algorithm could be further improved by formulating effective heuristics to guide search space exploration, which we leave as future work.

The last three columns in Figure 6 give information about the computed invariants. The column labeled " Invariants" gives the number of non-trivial loop invariants required to

[2] On the 26 benchmarks taken from external sources, HOLA has a success rate of 96.2% while BLAST, InvGen, and Interproc have success rates of 57.7%, 65.4%, and 50% respectively.

[3] While some of these benchmarks do not require disjunctive invariants, Interproc still fails due to widening.

Name	Strengthenings	Iterations	Backtracks	# Invariants	Disjunctive?	Avg. Inv. Size
Benchmark 1	2	3	0	1	no	2.0
Benchmark 2	3	4	0	1	no	5.0
Benchmark 3	0	1	0	3	yes	1.7
Benchmark 4	1	6	4	1	yes	2.0
Benchmark 5	2	3	0	1	no	2.0
Benchmark 6	4	5	0	2	yes	5.0
Benchmark 7	2	14	11	1	yes	4.0
Benchmark 8	3	14	10	1	no	3.0
Benchmark 9	6	8	0	4	no	1.0
Benchmark 10	2	3	1	1	yes	5.0
Benchmark 11	2	14	12	1	yes	4.0
Benchmark 12	7	74	68	2	yes	5.0
Benchmark 13	2	28	25	1	yes	8.0
Benchmark 14	2	29	26	1	no	5.0
Benchmark 15	–	–	–	–	–	–
Benchmark 16	1	23	21	1	yes	2.0
Benchmark 17	7	7	2	2	no	2.0
Benchmark 18	2	234	226	1	no	4.0
Benchmark 19	–	–	–	–	–	–
Benchmark 20	2	132	129	1	yes	1.5
Benchmark 21	2	3	0	1	yes	9.0
Benchmark 22	2	3	0	1	no	4.0
Benchmark 23	2	3	0	1	no	3.0
Benchmark 24	2	3	0	2	no	1.7
Benchmark 25	4	5	0	2	no	2.0
Benchmark 26	6	8	0	2	yes	5.0
Benchmark 27	7	8	0	3	yes	2.7
Benchmark 28	2	7	3	2	no	2.0
Benchmark 29	2	3	0	2	no	1.0
Benchmark 30	2	3	0	1	no	2.0
Benchmark 31	2	4	0	3	yes	5.3
Benchmark 32	3	27	25	1	yes	11.0
Benchmark 33	6	7	0	3	no	1.7
Benchmark 34	–	–	–	–	–	–
Benchmark 35	1	19	17	1	yes	3.0
Benchmark 36	2	7	4	4	no	3.5
Benchmark 37	2	94	92	1	yes	2.0
Benchmark 38	2	17	14	1	yes	4.0
Benchmark 39	0	1	0	1	yes	11.0
Benchmark 40	7	57	52	2	yes	3.0
Benchmark 41	3	4	0	1	yes	10.0
Benchmark 42	2	3	0	1	yes	5.0
Benchmark 43	1	2	0	1	no	3.0
Benchmark 44	1	46	44	1	yes	6.0
Benchmark 45	10	11	0	3	yes	6.0
Benchmark 46	2	10	7	1	yes	4.0

Figure 6. Statistics about algorithm and the benchmarks

verify the program. The column labaled "Disjunctive?" indicates whether the invariants computed by our algorithm involve disjunctions, and the last column indicates the average size of the computed invariants, measured in terms of the number of boolean connectives. As this data shows, many of these benchmarks involve multiple non-trivial invariants, often involving disjunctions.

7. Related Work

7.1 Other Techniques for Loop Invariant Generation

Existing techniques for loop invariant generation include abstract interpretation [1–4, 34], counterexample guided abstraction refinement (CEGAR) [7, 8, 35], constraint-based methods [5, 6, 36], guess-and-check approaches [37, 38], and techniques based on Craig interpolation [9–11, 39].

One dimension along which loop invariant generation techniques can be characterized is lazy vs. eager approaches. Eager techniques, such as abstract interpretation and constraint-based methods, compute all possible invariants they can about the program, while lazy techniques compute only those invariants that are necessary for showing the program's correctness. Similar to CEGAR and interpolation-based approaches, our technique is lazy: We strengthen loop invariants in a demand-driven way only when stronger invariants are needed to verify the correctness of the program.

Another dimension in which invariant generation techniques can be classified is whether they infer invariants of a predefined syntactic shape, such as only equalities or conjunctions. For example, in abstract interpretation, the choice of the abstract domain fixes the shape of the invariants to be inferred, such as octagons or polyhedra. Constraint-based approaches also fix a template (such as $\alpha x + \beta y \leq \gamma$) which syntactically restricts the class of invariants that can be inferred. In contrast, the approach proposed in this paper does not syntactically restrict the class of invariants to be inferred and can therefore discover linear invariants with arbitrary boolean structure.

Similar to many guess-and-check, CEGAR, and interpolation approaches, our technique generates candidate invariants whose correctness must later be checked. For example, Houdini [37] uses syntactic clues to guess candidate invariants, and Daikon [38] utilizes observed run-time values to generate guesses. CEGAR and interpolation-based approaches generate candidate invariants from counterexample traces in order to rule out at least the observed spurious trace. Our technique differs from all of these approaches in that our candidate invariants are, by construction, always sufficient to show the correctness of some assertion in the program. In this sense, our technique is more goal directed than approaches that speculate invariants. Similar to interpolation-based techniques, our approach is completely semantic and uses logical inference to compute candidate invariants. However, a difference is that interpolation approaches generate invariants that are implied by underapproximations of the reachable states, whereas here we speculate invariants that are consistent with an overapproximation of the reachable states.

The loop invariant generation method presented in [40] bears similarities to our technique. Like our approach, [40] also starts from assertions to be proven and generates inductive invariants by iteratively strengthening an initial candidate invariant. The main difference between our method and [40] is the strengthening mechanism: Here, the strengthening is performed by applying abductive inference on invalid VCs. In contrast, the method of [40] repeatedly computes the weakest precondition of the assertion through the loop body and performs generalization by dropping predicates that are not shared across loop iterations. Furthermore, unlike our approach, the inferred strengthenings may not be sufficient to fix the VC even if they are inductive. Finally, since [40] is based on symbolic execution, it relies on exact loop preconditions which may be overapproximate in our setting.

Another approach related to the technique considered here is IC3 [22, 41, 42]. As in interpolation methods, inference in IC3 is geared to rule out counterexamples, which is not the case here. There is an interesting connection to our method, however. IC3 starts with a known (time-bounded) fact ϕ_1 and infers a ϕ_2 that is inductive *relative* to ϕ_1. Here, we do the opposite. We start with a conjecture ϕ_2 and infer by abduction a new conjecture ϕ_1, such that ϕ_2 is inductive relative to ϕ_1. A difference is that IC3 conservatively infers time-bounded facts, while we always speculate that our abductive inferences are invariant, so we may have to backtrack.

Some loop invariant generation techniques, such as abstract interpretation and constraint-based methods, are guaranteed to terminate, while others, such as CEGAR, can diverge. Similar to CEGAR, our technique also does not have termination guarantees: While we consider a finite number of abductive strengthenings at each step, there is no bound on the number of possible strengthenings; as a result, the algorithm as presented in this paper may diverge in an infinite chain of speculations. However, it is easily possible to modify our algorithm to guarantee termination, for example, by limiting the number of abductive strengthenings that may be performed.

7.2 Use of Abduction in Program Verification

The concept of logical abduction, which was originally proposed by Peirce [15], has found a number of useful applications in program verification. Specifically, abduction has been used in modular shape analysis based on separation logic [43], for inferring missing preconditions in the analysis of logic programs [44], and for constructing underapproximations in quantified logical domains [45]. All these techniques use different algorithms for performing abduction than the one we consider here and apply logical abduction in the context of very different problem domains.

To the best of our knowledge, the only previous application of abduction to invariant generation is in the context of resource invariant synthesis using separation logic [46]. For each lock in the program, a *resource invariant* is an assertion that holds whenever no thread has acquired that lock. The work described in [46] uses bi-abductive inference in separation logic to infer such resource invariants. There are several key differences between the work described in [46] and this paper. First, here, we consider the problem of inferring Presburger arithmetic invariants whereas [46] considers resource invariants expressible in separation logic. However, the present work cannot be viewed as simply applying that method to arithmetic invariants. Among many differences, [46] uses a fixed heap abstration function to compute bi-abductive inferences and generates only one refinement for a given counterexample. Using a fixed abstraction would not work for our application domain because the heart of our approach is to generate a range of abductive inferences and construct the invariant by backtracking search.

Our own recent work has applied abductive inference to the diagnosis of error reports generated by verification tools [47] and the construction of circular compositional program proofs [48]. Specifically, [47] uses abductive inference to generate queries that are used to help users decide whether static analysis warnings correspond to real bugs or false alarms. Our recent work described in [48] uses abduction to decompose the program's proof of correctness into small local lemmas, each of which are proven by a different tool or abstraction in a circular compositional manner. The abduction algorithm we present in Section 4.2 is similar to the abduction algorithms used in [47, 48], but adapted for the purpose of generating candidate strengthenings. The main contribution of the present paper is to apply logical abduction in the context of automatic numeric invariant generation. To the best of our knowledge, this is the first paper to demonstrate that a logical abduction procedure based on quantifier elimination is powerful for automatically inferring interesting numeric loop invariants.

8. Conclusion and Future Work

In this paper, we have presented a new method for generating loop invariants that are expressible in Presburger arithmetic. Our technique performs a backtracking search that combines Hoare-style program reasoning with logical abduction based on quantifier elimination to speculate candidate invariants. The inferred loop invariants are iteratively strengthened until they are both inductive and strong enough to prove program correctness. Experiments on a set of benchmarks taken from a variety of existing and new sources indicate that our approach is effective in practice and can infer invariants that cannot be established by existing tools.

In future work, we plan extend the applicability of the approach described in this paper by addressing two key issues:

1. **Scalability:** To make the proposed approach useful for analyzing large, real-world programs, we believe it is necessary to reduce backtracking as much as possible. For this purpose, we plan to explore different search strategies than the simple depth-first strategy considered in this paper. Furthermore, we plan to use underapproximations for quickly ruling out wrong abductive speculations.
2. **Abduction in richer logical theories:** While this paper only addresses numeric invariant generation, making the proposed approach practical for real-world programs requires reasoning about data structure invariants. Since such invariants are only expressible in richer logical theories such as the theory of uninterpreted functions or the theory of arrays, we plan to explore abduction algorithms for richer logics. Since such logics do not admit quantifier elimination, abduction must be performed either through sound, but approximate quantifier elimination procedures such as [49] or through domain-specific inference rules for a particular theory.

References

[1] Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: POPL, ACM (1978) 84–96

[2] Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1) (2006) 31–100

[3] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, ACM (1979) 269–282

[4] Karr, M.: Affine relationships among variables of a program. A.I. (1976) 133–151

[5] Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: Computer Aided Verification, Springer (2009) 634–640

[6] Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Computer Aided Verification, Springer (2003) 420–432

[7] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: International conference on Model checking software. (2003) 235–239

[8] Ball, T., Rajamani, S.: The slam toolkit. In: Computer aided verification, Springer (2001) 260–264

[9] McMillan, K.: Lazy annotation for program testing and verification. In: Computer Aided Verification, Springer (2010) 104–118

[10] Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. ACM SIGPLAN Notices **39**(1) (2004) 232–244

[11] McMillan, K.: Interpolation and sat-based model checking. In: Computer Aided Verification, Springer (2003) 1–13

[12] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Pro-

gramming language design and implementation. PLDI '02, New York, NY, USA, ACM (2002) 234–245

[13] Leino, K.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning, Springer (2010) 348–370

[14] Barnett, M., yuh Evan Chang, B., Deline, R., Jacobs, B., Leino, K.R.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science, Springer (2006) 364–387

[15] Peirce, C.: Collected papers of Charles Sanders Peirce. Belknap Press (1932)

[16] Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum satisfying assignments for SMT, CAV (2012)

[17] Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence **7**(91-99) (1972) 300

[18] Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language. Stanford University Technical Report

[19] Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. SAS (2011)

[20] Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In: CAV. (2009)

[21] Jeannet, B.: Interproc analyzer for recursive programs with numerical variables. http://pop-art. inrialpes. fr/interproc/interprocweb. cgi

[22] Bradley, A.: Understanding IC3. Theory and Applications of Satisfiability Testing–SAT 2012 (2012) 1–14

[23] Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. Volume 43., ACM (2008) 281–292

[24] Jhala, R., McMillan, K.: A practical and complete approach to predicate refinement. Tools and Algorithms for the Construction and Analysis of Systems (2006) 459–473

[25] Sharma, R., Nori, A., Aiken, A.: Interpolants as classifiers. In: Computer Aided Verification, Springer (2012) 71–87

[26] Gulavani, B., Rajamani, S.: Counterexample driven refinement for abstract interpretation. Tools and Algorithms for the Construction and Analysis of Systems (2006) 474–488

[27] Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: ACM SIGPLAN Notices. Volume 42., ACM (2007) 277–289

[28] Gulavani, B., Chakraborty, S., Nori, A., Rajamani, S.: Automatically refining abstract interpretations. TACAS (2008) 443–458

[29] Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. PLDI '07, New York, NY, USA, ACM (2007) 300–309

[30] Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Form. Asp. Comput. **20**(4-5) (June 2008) 379–405

[31] Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. SIGSOFT '06/FSE-14, New York, NY, USA, ACM (2006) 117–127

[32] http://pub.ist.ac.at/ agupta/invgen/: InvGen tool

[33] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php: NECLABS NECLA verification benchmarks

[34] Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: Verification, Model Checking, and Abstract Interpretation, Springer (2009) 229–244

[35] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (September 2003) 752–794

[36] Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Verification, Model Checking, and Abstract Interpretation, Springer (2009) 120–135

[37] Flanagan, C., Leino, K.: Houdini, an annotation assistant for esc/java. FME 2001: Formal Methods for Increasing Software Productivity (2001) 500–517

[38] Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1-3) (2007) 35–45

[39] McMillan, K.: Lazy abstraction with interpolants. In: Computer Aided Verification, Springer (2006) 123–136

[40] Păsăreanu, C.S., Visser, W.: Verification of java programs using symbolic execution and invariant generation. In: SPIN Workshop on Model Checking Software. Springer (2004) 164–181

[41] Bradley, A.: Sat-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation, Springer (2011) 70–87

[42] Somenzi, F., Bradley, A.: IC3: where monolithic and incremental meet. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD Inc (2011) 3–8

[43] Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL **44**(1) (2009) 289–300

[44] Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proceedings of the 1994 International Symposium on Logic programming, Citeseer (1994) 377–391

[45] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246

[46] Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: Proceedings of the 7th Asian Symposium on Programming Languages and Systems. APLAS '09, Berlin, Heidelberg, Springer-Verlag (2009) 259–274

[47] Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. PLDI '12, New York, NY, USA, ACM (2012) 181–192

[48] Li, B., Dillig, I., Dillig, T., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'13, Springer-Verlag (2013) 370–384

[49] Gulwani, S., Musuvathi, M.: Cover Algorithms. In: ESOP. (2008) 193–207