3-2017

FiB: Squeezing loop invariants by interpolation between forward/backward predicate transformers

Shang-Wei LIN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Hao XIAO

Yang LIU

David SANA

See next page for additional authors

F*i*B: Squeezing Loop Invariants by Interpolation between Forward/Backward Predicate Transformers

Shang-Wei Lin*, Jun Sun†, Hao Xiao*, Yang Liu*, David Sanán*, and Henri Hansen‡

*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†Singapore University of Technology and Design, Singapore
‡Tampere University of Technology, Finland

***Abstract*—Loop invariant generation is a fundamental problem in program analysis and verification. In this work, we propose a new approach to automatically constructing inductive loop invariants. The key idea is to aggressively squeeze an inductive invariant based on Craig interpolants between forward and backward reachability analysis. We have evaluated our approach by a set of loop benchmarks, and experimental results show that our approach is promising.**

I. Introduction

In program verification, loops are challenging to handle. One way to prove that a loop satisfies its postcondition under a precondition is based on *inductive loop invariants*. Intuitively, an inductive loop invariant is a property which holds in each iteration of the loop. Given an inductive loop invariant, as long as it is weaker than the precondition, and its conjunction with the negation of the loop condition is stronger than the postcondition, we can conclude that the loop establishes its postcondition if it terminates. However, program verification based on loop invariants does not come for free. The key challenge is how to construct inductive loop invariants automatically, which is a fundamental problem in program analysis and verification.

Given a loop with precondition P and postcondition Q, traditional forward analysis [23], [19] tries to obtain the reachability of the loop after each iteration, as shown in Fig. 1 (a). Let $F_t(P)$ be the set of forward reachable states starting from the precondition P after the t-th iteration of the loop, where $F_0(P) = P$. Notice that $F_i(P) \implies F_{i+1}(P)$ for all $i \geq 0$, does not necessarily hold if we consider the strongest condition after the loop is executed (c.f. Section III for more details). Assume that the loop terminates after t iterations for some $t \geq 0$, i.e., $F_t(P)$ does not satisfy the loop condition. If $F_t(P) \implies Q$, then we can conclude that the loop satisfies its postcondition after it terminates. However, this approach may not terminate in general, or it may take a large number of forward iterations to find such a $F_t(P)$ for some $t \geq 0$.

Similarly, traditional backward reachability analysis [23], [19] tries to obtain the set of backward reachable states, denoted by $B_t(Q)$, from the postcondition Q after t iterations (assume the loop condition is still satisfied). Notice that $B_i(Q) \implies B_{i+1}(Q)$ for all $i \geq 0$ does hold if we consider the weakest condition after the loop is executed backward (c.f. Section III for more details). If we can find a backward reachability $B_t(Q)$ for some $t \geq 0$ such that $P \implies B_t(Q)$, then we conclude that the loop satisfies its postcondition, as shown in Fig. 1 (b). However, this backward approach may not terminate in general as well, or it may take a large number of backward iterations to find such a $B_t(Q)$ for some $t \geq 0$.

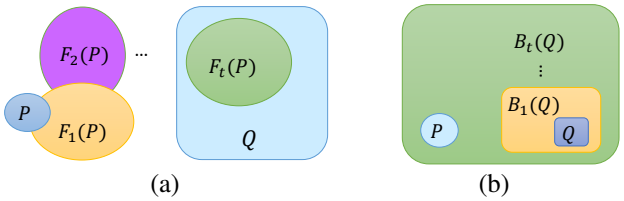

Fig. 1. Traditional (a) Forward and (b) Backward Reachability

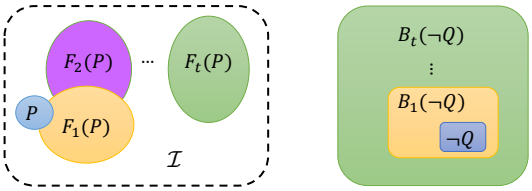

Fig. 2. Our Approach

In this paper[1], we propose an approach to automatically constructing inductive loop invariants in the form of Boolean combinations of linear integer constraints over program variables. Our approach is to squeeze an invariant between forward and backward reachability of the loop. Although the forward and backward approaches may not terminate in general, they do provide some hints to construct inductive invariants. The intuitive idea behind our approach is as follows. Instead of starting from the postcondition Q, we perform the backward reachability analysis from $\neg Q$, the negation of the postcondition. If the loop does have an inductive invariant $\mathcal{I}$ to establish the postcondition Q, as shown in Fig. 2, then $\cup_{t=0}^{\infty} F_t(P)$ will not intersect with $B_{t\to\infty}(\neg Q)$; otherwise $\mathcal{I}$ would not be inductive. In addition, $\cup_{t=0}^{\infty} F_t(P)$ will be included in the inductive invariant, i.e., $\cup_{t=0}^{\infty} F_t(P) \implies \mathcal{I}$. Furthermore, $\mathcal{I}$ will not intersect with $B_{t\to\infty}(\neg Q)$ as well; otherwise, $\mathcal{I}$ would not be an invariant to establish Q. Based on the above observation, an inductive invariant $\mathcal{I}$ is actually

[1]The corresponding author, Shang-Wei Lin, can be contacted via the following e-mail address: shang-wei.lin@ntu.edu.sg.

978-1-5386-2684-9/17$15.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

a Craig interpolant [18] (c.f. Definition 2) for $\cup_{i=0}^{t} F_i(P)$ with respect to $B_t(\neg Q)$. That is, we can squeeze an inductive invariant between the forward reachability from P and the backward reachability from $\neg Q$ by obtaining an interpolant in between. If the number of forward/backward steps is sufficient, an inductive invariant in between should be squeezed out. This observation gives a systematic way to construct an inductive invariant. The details of our approach is described in Section IV. To summarize, this work makes the following contributions:

- We propose a novel approach to automatically constructing inductive loop invariants for loop structures consisting of multiple paths inside. The generated invariant is in the form of Boolean combinations of linear integer constraints over program variables. The proposed approach squeezes loop invariants between forward and backward analysis by obtaining interpolants in between. To the best of our knowledge, this is the first work to combine interpolation techniques with forward and backward predicate transformers in Hoare Logic [34].
- We have implemented our approach in a tool, called F*i*B. We compared F*i*B with other state-of-the-art invariant generation tools: BLAST [33], InvGen [31], Interproc [35], CPAchecker [9], ITP[41], and HOLA[20]. Our experimental results show that our approach is promising for advancing the state-of-the-art invariant generation for numeric loops.

The rest of this paper is organized as follows. Section II illustrates our approach with motivating examples. Section III reviews preliminary backgrounds. The details of the proposed approach is introduced in Section IV. Evaluations of our approach are presented in Section V. Section VI summarizes related works, and Section VII concludes this work.

II. Motivating Examples

In this section, we illustrate the intuitive idea behind our approach using two example programs. The first example is shown in Fig. 3 (a). The precondition of the loop is $P : (x = 0 \wedge y = 0)$, and the postcondition (assertion) is $Q : y = 100$. Traditional forward analysis based on strongest postcondition requires to execute 100 iterations of the loop to reach $F_{100}(P) : (x = 100 \wedge y = 100)$. Since $F_{100}(P) \wedge (x = 100) \implies Q$, we can conclude that the assertion Q is satisfied if the loop terminates.

Similarly, traditional backward analysis based on weakest precondition also requires 100 iteration to find $B_{100}(Q) : \bigvee_{i=0}^{100}(x = i \wedge y = i)$. Since $P \implies B_{100}(Q)$, we can conclude that the assertion is satisfied if the loop terminates. If we change the constant from 100 to 100000, traditional approaches require a huge number of iterations to prove the assertion. Fig. 3 (b) shows the second example, which is almost the same as the first one except that the constant 100 is replaced by a variable $n \geq 0$. In this example, traditional (either forward or backward) approaches do not terminate because the bound is now a variable instead of a constant.

```
assume(x == 0);
assume(y == 0);

while(x != 100){
   x++; y++;
}

assert(y == 100);
```

(a)

```
assume(x == 0);
assume(y == 0);
assume(n >= 0);

while(x != n){
   x++; y++;
}
assert(y == n);
```

(b)

Fig. 3. Simple Examples

$$
\begin{array}{rcl}
\mathsf{Stmt} & \triangleq & \mathsf{skip} \mid \mathsf{Stmt}; \mathsf{Stmt} \mid x := \mathsf{Exp} \mid x := \texttt{nondet} \\
 & & \mid \texttt{if}\ \mathsf{BExp}\ \texttt{then}\ \mathsf{Stmt}\ \texttt{else}\ \mathsf{Stmt} \\
\mathsf{Exp} & \triangleq & n \mid x \mid \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp} \\
 & & \mid \mathsf{Exp} * n \mid \mathsf{Exp}\ \%\ n \\
\mathsf{BExp} & \triangleq & \mathsf{False} \mid b \mid \texttt{nondet} \mid \neg \mathsf{BExp} \mid \mathsf{BExp} \wedge \mathsf{BExp} \\
 & & \mid \mathsf{Exp} < \mathsf{Exp} \mid \mathsf{Exp} = \mathsf{Exp}
\end{array}
$$

Fig. 4. Syntax of IMP

Our approach works as follows for the second program. Let us first consider the case where the loop executes without any iteration. In this case, the forward reachability $F_0(P)$ is $P : (x = 0 \wedge y = 0 \wedge n \geq 0)$. Then, we perform the backward analysis. Notice that unlike traditional backward approach starting from the postcondition Q, we obtain backward reachability from the negation of the postcondition, i.e., $\neg Q$. If the loop executes backward without any iteration from $\neg Q$, the backward reachability $B_0(\neg Q)$ would be the conjunction of $\neg Q$ and the negation of the loop condition, i.e., $(y \neq n \wedge x = n)$. Then, we use an SMT solver to check the satisfiability of $F_0(P) \wedge B_0(\neg Q)$. That is, we check whether the loop violates its postcondition in zero iteration. Obviously, the formula is not satisfiable, and we can obtain an interpolant $\mathcal{I} : (x = y)$, c.f. Definition 2, for $F_0(P)$ with respect to $B_0(\neg Q)$ from the SMT solver. Intuitively, the interpolant $\mathcal{I}$ is an abstraction of $F_0(P)$, i.e., $F_0(P) \implies \mathcal{I}$, and $\mathcal{I}$ is still inconsistent with $B_0(\neg Q)$. Then, we check whether the interpolant $\mathcal{I}$ is inductive, i.e., if $\mathcal{I}$ holds initially in the loop, no matter how many iterations the loop executes, $\mathcal{I}$ still holds. The formula for the inductiveness checking is $(x = y) \wedge (x' = x+1) \wedge (y' = y+1) \wedge (x \neq n) \implies (x' = y')$. We can check the validity of this formula by an SMT solver as well (c.f. Sections III and IV), and $(x = y)$ is inductive and strong enough to prove the postcondition Q. Thus, $(x = y)$ is the loop invariant obtained by our approach. However, if $\mathcal{I}$ is not inductive, we consider the second case where the loop executes for one iteration. The same procedure is performed: (1) check the satisfiability of $(F_0(P) \vee F_1(P)) \wedge B_1(\neg Q)$, (2) obtain an interpolant $\mathcal{I}$ if the formula is not satisfiable, and (3) check the inductiveness of $\mathcal{I}$. We keep increasing the number of iterations and performing the same procedure until we find an inductive invariant. The details of our approach is introduced in Section IV.

$$
\begin{array}{rcl}
\mathsf{wp}(\mathsf{skip}, Q) & \Leftrightarrow & Q \\
\mathsf{wp}(x := e, Q) & \Leftrightarrow & Q[x \mapsto e] \\
\mathsf{wp}(x := \texttt{nondet}, Q) & \Leftrightarrow & \forall x \colon Q \\
\mathsf{wp}(\mathsf{s}_1; \mathsf{s}_2, Q) & \Leftrightarrow & \mathsf{wp}(\mathsf{s}_1, \mathsf{wp}(\mathsf{s}_2, Q)) \\
\mathsf{wp}(\texttt{if } b \texttt{ then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2, Q) & \Leftrightarrow & (b \wedge \mathsf{wp}(\mathsf{s}_1, Q)) \vee \\
 & & (\neg b \wedge \mathsf{wp}(\mathsf{s}_2, Q)) \\
\mathsf{wp}(\texttt{if nondet then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2, Q) & \Leftrightarrow & \mathsf{wp}(\mathsf{s}_1, Q) \vee \\
 & & \mathsf{wp}(\mathsf{s}_2, Q)
\end{array}
$$

Fig. 5. Rules for Weakest Precondition

$$
\begin{array}{rcl}
\mathsf{sp}(P, \mathsf{skip}) & \Leftrightarrow & P \\
\mathsf{sp}(P, x := e) & \Leftrightarrow & \exists x_0 \colon P[x \mapsto x_0] \wedge \\
 & & x = e[x \mapsto x_0] \\
\mathsf{sp}(P, x := \texttt{nondet}) & \Leftrightarrow & \exists x_0 \colon P[x \mapsto x_0] \\
\mathsf{sp}(P, \mathsf{s}_1; \mathsf{s}_2) & \Leftrightarrow & \mathsf{sp}(\mathsf{sp}(P, s_1), s_2) \\
\mathsf{sp}(P, \texttt{if } b \texttt{ then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2) & \Leftrightarrow & \mathsf{sp}(P \wedge b, \mathsf{s}_1) \vee \\
 & & \mathsf{sp}(P \wedge \neg b, \mathsf{s}_2) \\
\mathsf{sp}(P, \texttt{if nondet then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2) & \Leftrightarrow & \mathsf{sp}(P, \mathsf{s}_1) \vee \mathsf{sp}(P, \mathsf{s}_2)
\end{array}
$$

Fig. 6. Rules for Strongest Postcondition

III. Background

In this work, we consider the imperative (IMP) language, whose syntax is shown in Fig. 4. In the IMP language, programs consist of one or more statements. Statements, denoted by the symbol Stmt, include skip, sequencing, assignment, and conditional statements. The keyword `nondet` denotes an arbitrary value in the type of the assigned variable. The IMP language supports two basic types: Booleans and integers. There are two kinds of expressions in IMP language, namely integer expressions (denoted by the Exp symbol) and Boolean expressions (denoted by the BExp symbol). A term in Exp is of the type of integers where x is an integer variable and n is an integer constant, while a term in BExp is of Boolean type where b is a Boolean variable.

To reason about the correctness of an IMP program, we use the interpretation of Hoare Logic [34]. A Hoare triple is a formula of the form $\{P\}\,\mathsf{s}\,\{Q\}$, where s is a Stmt formula representing a statement in an IMP program, and P and Q are the *precondition* and *postcondition* of the statement s, respectively. The Hoare triple $\{P\}\,\mathsf{s}\,\{Q\}$ is *partially correct*[2] if the statement s is executed in a state in which P holds, and then it terminates in a state in which Q holds unless it aborts or runs forever.

In this work, we consider two predicate transformers [23], [19]: wp and sp. The function wp takes as inputs an IMP statement s and a postcondition Q and returns the *weakest precondition* of s with respect to Q, denoted by $\mathsf{wp}(\mathsf{s}, Q)$. The function sp takes as inputs an IMP statement and a precondition P and returns the *strongest postcondition* of s with respect to P, denoted by $\mathsf{sp}(P, \mathsf{s})$. The rules to calculate the weakest precondition and strongest postcondition for each primitive statement are given in Fig. 5 and Fig. 6, respectively [53]. A Hoare triple $\{P\}\,\mathsf{s}\,\{Q\}$ can be proved if either $P \implies \mathsf{wp}(\mathsf{s}, Q)$ holds, or $\mathsf{sp}(P, \mathsf{s}) \implies Q$ holds.

The target loop structure in this work is a single loop with multiple paths, which can be represented as an *annotated loop* of the form: $\{P\}$ `while` κ `do` S `done` $\{Q\}$. The BExp formula κ is the *loop guard*. The loop body S is a Stmt formula representing a sequence of statements. The BExp formulas P and Q are the *precondition* and *postcondition* of the annotated loop, respectively.

One way to validate the Hoare triple for an annotated loop is based on the sp predicate transformer, in which we need to prove that $\mathsf{sp}(P, \texttt{while } \kappa \texttt{ do S done}) \implies Q$. Traditionally [23], [19], it is done by approximating the strongest postcondition of the loop. We can define the strongest postcondition of a loop in a recursive way as follows:

$$
\begin{array}{c}
\mathsf{sp}(P, \texttt{while } \kappa \texttt{ do S done}) \\
\Updownarrow \\
(P \wedge \neg\kappa) \vee \mathsf{sp}(\mathsf{sp}(\mathsf{P} \wedge \kappa, \mathsf{S}), \texttt{while } \kappa \texttt{ do S done})
\end{array}
$$

However, the above recursive construction of the strongest postcondition may not terminate in general, which is not practical to validate an annotated loop. Furthermore, even if it terminates, it may take a large number of iterations to converge to a fix-point.

Another way to validate an annotated loop is based on the wp predicate transformer [23], [19], in which we need to prove that $P \implies \mathsf{wp}(\texttt{while } \kappa \texttt{ do S done}, Q)$. Similarly, We can approximate the weakest precondition of the loop by considering the number of iterations required to establish the postcondition Q. Let B_t be a predicate describing the set of states from which the loop terminates within t iterations and establish Q. We can define B_t in a recursive way as follows, where $B_0 = (\neg\kappa \wedge Q)$:

$$B_t = B_0 \vee (\kappa \wedge \mathsf{wp}(\mathsf{S}, B_{t-1}))$$

It means that to establish Q within t iterations, the loop can either terminate immediately without any iteration and then establish Q, or perform one iteration and reach a state where it terminates within $t-1$ iterations and then establish Q. Theoretically, $\mathsf{wp}(\texttt{while } \kappa \texttt{ do S done}, Q)$ is equivalent to $\lim_{k\to\infty} B_t$. Since B_t is an under-approximation of $\mathsf{wp}(\texttt{while } \kappa \texttt{ do S done}, Q)$ for all $t \geq 0$, in practice, as long as we can find B_t for some $t \geq 0$ such that $P \implies B_t$ holds, then we can conclude that $P \implies \mathsf{wp}(\texttt{while } \kappa \texttt{ do S done}, Q)$ holds. However, this approach may not terminate in general, or it may take a large number of iterations to find such a B_t for some $t \geq 0$.

[2]Total correctness requires that the statement s has to terminate. We only consider partial correctness because the target statement in this work is a loop which may not terminate in general.

A promising way to validate an annotated loop is based on loop invariants, as formulated in Definition 1. However, the main problem of validating an annotated loop based on loop invariants is how to generate inductive loop invariants automatically. In the next section, we are going to introduce our approach to generating loop invariants automatically based on Craig interpolation, as formulated in Definition 2.

Definition 1: A *loop invariant* $\mathcal{I}$ of an annotated loop is a formula satisfying the following conditions: (1) $P \implies \mathcal{I}$, (2) $\mathcal{I} \wedge \neg\kappa \implies Q$, and (3) $\{\mathcal{I} \wedge \kappa\}\ \mathsf{S}\ \{\mathcal{I}\}$.

Definition 2: Given two Boolean formulas A and B such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* for A with respect to B is a formula $\hat{A}$ satisfying the following properties: (1) $\hat{A}$ is an abstraction of A, i.e., $A \implies \hat{A}$, (2) $\hat{A} \wedge B$ is unsatisfiable, and (3) $\hat{A}$ refers only to the common variables of A and B.

The Craig interpolation lemma [18] states that an interpolant always exists for inconsistent (unsatisfiable) formulas in first-oder logic. Modern SAT or SMT solvers, e.g., Z3 [3] and MathSAT [1], support Craig interpolation from unsatisfiable formulas.

IV. Squeezing Loop Invariants

In this section, we introduce how an invariant of a loop can be squeezed based on interpolation between forward/backward reachability analyses. We propose two approaches. The first one, introduced in Section IV-A is based on interpolation with respect to forward reachability. The second, introduced in Section IV-B, is based on interpolation with respect to backward reachability. Section IV-C discusses some extensions.

A. Forward Interpolation

Given an annotated loop, $\{P\}$ `while` κ `do` S `done` $\{Q\}$, we can consider its forward reachability with the precondition P by calculating its strongest postcondition. Let us use $A_i = \mathsf{sp}(A_{i-1} \wedge \kappa, \mathsf{S})$ for $i \geq 1$ to denote the strongest postcondition after the i-th iteration, where $A_0 = P$. If the loop does have an inductive invariant $\mathcal{I}$, then the reachability after each iteration must be contained in $\mathcal{I}$ (otherwise, $\mathcal{I}$ would not be inductive), i.e., the condition $\bigvee_{i=0}^{\infty} A_i \implies \mathcal{I}$ must hold.

On the other hand, we can also consider the backward reachability of the loop by calculating the weakest precondition. However, for the backward reachability, we start from $\neg Q$, the negation of the postcondition. Let $B_i = B_0 \vee (\kappa \wedge \mathsf{wp}(\mathsf{S}, B_{i-1}))$ for $i \geq 1$ be the predicate representing the set of states from which the loop terminates within i iterations and violates the postcondition Q, where $B_0 = (\neg Q \wedge \neg\kappa)$ represents that the loop violates its postcondition Q without performing any iterations. If the loop does have an inductive invariant $\mathcal{I}$ to establish its postcondition Q, then the formula $\mathcal{I} \wedge B_i$ must be unsatisfiable for all $i \geq 0$, otherwise $\mathcal{I}$ would not be inductive and establish Q.

Based on the above observations, we can find that an inductive invariant $\mathcal{I}$ (if it exists) of a loop is actually an interpolant for $(\bigvee_{i=0}^{\infty} A_i)$ with respect to B_∞ because $\bigvee_{i=0}^{\infty} A_i \implies \mathcal{I}$ and $\mathcal{I} \wedge B_\infty$ is unsatisfiable. Notice that the inverse is not true, i.e., the interpolant $\mathcal{I}$ is not necessary to be inductive. However, it does provide a way to find an inductive invariant of a loop. The basic idea is as follows. Firstly, we calculate, for t iterations, the forward reachability $(\bigvee_{i=0}^{t} A_i)$ as well as the backward reachability B_t. Secondly, we obtain an interpolant $\mathcal{I}$ for $(\bigvee_{i=0}^{t} A_i)$ with respect to B_t and check whether $\mathcal{I}$ is inductive or not. Thirdly, if $\mathcal{I}$ is an inductive invariant, then we are done. Otherwise, we increase the value of t and repeat the first and second steps until we find an inductive invariant. Algorithm 1 shows the pseudo-code of squeezing an inductive invariant of a loop by interpolation between the forward and backward reachabilities. The details of Algorithm 1 are described as follows:

- Initially, we set A_0 to be P, and B_0 to be $(\neg Q \wedge \neg\kappa)$, respectively (lines 1–2). And, we use t to denote the number of forward iterations, and r for the number of backward iterations. Their initial values are set to be zero, respectively (line 3). Notice that the values of t and r might become different during the following process.
- A decision procedure is performed to check whether the formula $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ is satisfiable or not (line 5). If the formula is not satisfiable, we can further obtain an interpolant $\mathcal{I}_t$ for $(\bigvee_{i=0}^{t} A_i)$ with respect to B_r. According to the characteristic of interpolants (c.f. Definition 2), we are guaranteed to have the following two properties: (1) $\bigvee_{i=0}^{t} A_i \implies \mathcal{I}_t$ and (2) $\mathcal{I}_t \wedge B_r$ is not satisfiable. That is, the forward reachability for t iterations from the precondition P is included in $\mathcal{I}_t$, and $\mathcal{I}_t$ is not going to violate the postcondition Q in r iterations, as illustrated in Fig. 7 (a). Then, we check whether $\mathcal{I}_t$ is inductive. If yes, we are done, and the

Algorithm 1: Squeeze Invariant – Forward

input : An annotated loop: $\{P\}$ `while` κ `do` S `done` $\{Q\}$
output: (yes/no, $\mathcal{I}$), where $\mathcal{I}$ is a loop invariant

```
1  A_0 ⟵ P ;
2  B_0 ⟵ (¬κ ∧ ¬Q) ;
3  t ⟵ r ⟵ 0 ;
4  while True do
5      if (⋁_{i=0}^{t} A_i) ∧ B_r is not satisfiable then
6          Let I_t be the interpolant for (⋁_{i=0}^{t} A_t) w.r.t. B_r ;
7          if {I_t ∧ κ} S {I_t} then return (yes, I_t) ;
8          A_{t+1} ⟵ sp(I_t ∧ κ, S) ;
9          t ⟵ t + 1 ;
10         B_{r+1} ⟵ B_0 ∨ (κ ∧ wp(S, B_r)) ;
11         r ⟵ r + 1 ;
12     else
13         if A_t is concrete then return (no, ⊥) ;
14         else
15             while A_t is not concrete do t ⟵ t − 1 ;
16             A_{t+1} ⟵ sp(A_t ∧ κ, S) ;
17             t ⟵ t + 1 ;
```

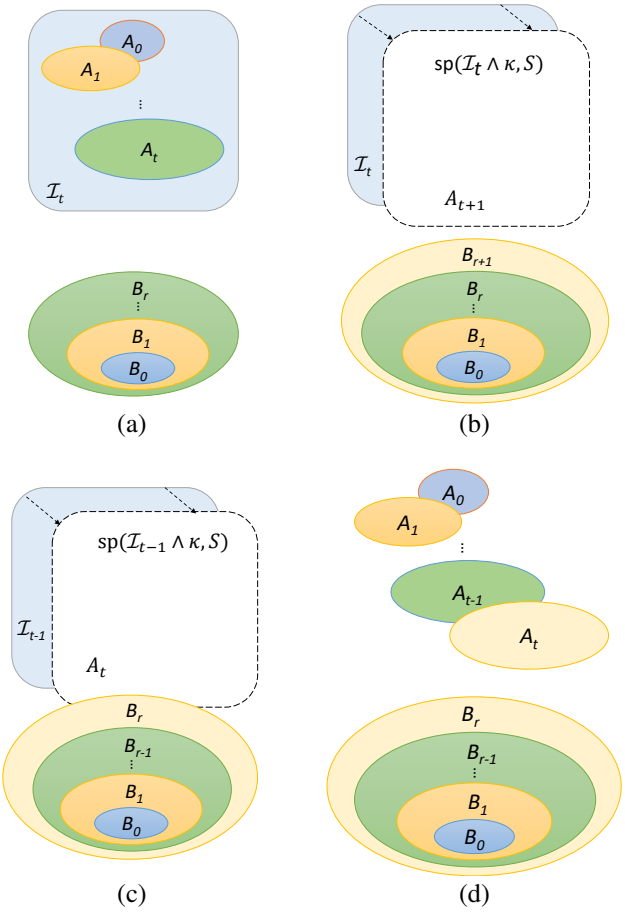

Fig. 7. Squeezing Invariants

inductive invariant $\mathcal{I}_t$ is returned (line 7). If $\mathcal{I}_t$ is not inductive, we advance forward for one more iteration based on $\mathcal{I}_t$ (lines 8–9). The reason for advancing forward from $\mathcal{I}_t$ is that we may converge toward an inductive invariant more quickly than from the concrete reachability $(\bigvee_{i=0}^{t} A_i)$ because $\mathcal{I}_t$ is an over-approximation of the concrete reachability, as illustrated in Fig. 7 (b). Similarly, for backward reachability, we advance backward for one more iteration from B_r (lines 10–11).

- Notice that for the sequence of forward reachability $A_0, A_1, \ldots, A_t$, if there exists $0 < j \leq t$ such that A_j is an over-approximation (i.e., A_j is obtained by the strongest postcondition operator from the interpolant $\mathcal{I}_{j-1}$ instead of from the concrete reachability A_{j-1}, as shown in Fig. 7 (b), then every A_m for $j \leq m \leq t$ would be an over-approximation as well. Thus, if the formula $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ is satisfiable (line 12), there could be two cases. The first case is that A_t is concrete, which means that $A_0, A_1, \ldots, A_t$ are all concrete. In this case, we conclude that the loop violates its postcondition Q because the solution to the satisfiability problem of $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ is such a counterexample starting from P and violating Q within $t + r$ iterations (line 13). The second case is shown in Fig. 7 (c), where A_t is an over-approximation. We cannot conclude anything for this case because A_t may contain spurious counterexamples. To be sound and simplify the process, we backtrack to the latest concrete forward reachability (line 15), from which we advance forward for one more iteration (lines 16–17), as illustrated in Fig. 7 (d). After that, the new forward and backward reachability layout will be analyzed in the next iteration of Algorithm 1. We repeat the process until either a counterexample or an inductive invariant is found.

The soundness of Algorithm 1 is proved by Lemma 1 and Theorem 1. Notice that Algorithm 1 does not guarantee its termination. In practice, we can set an upper bound on the number of forward or backward iterations.

Lemma 1: If $A \wedge (B \vee C)$ is unsatisfiable, then $A \wedge B$ is also unsatisfiable.

Theorem 1: Algorithm 1 is correct.

Proof 1: Given a loop: $\{P\}$ `while` κ `do` S `done` $\{Q\}$, we want to prove that the invariant $\mathcal{I}_t$ returned by Algorithm 1 satisfies the following three conditions: (1) $P \implies \mathcal{I}_t$, (2) $\mathcal{I}_t \wedge \neg\kappa \implies Q$, and (3) $\{\mathcal{I}_t \wedge \kappa\}$ S $\{\mathcal{I}_t\}$. Since $\mathcal{I}_t$ is returned by Algorithm 1 only if condition (3) holds, we only have to prove conditions (1) and (2).

Condition (1): we know that $P = A_0 \implies (\bigvee_{i=0}^{t} A_i)$, and $(\bigvee_{i=0}^{t} A_i) \implies \mathcal{I}_t$ because $\mathcal{I}_t$ is an interpolant for $(\bigvee_{i=0}^{t} A_i)$ with respect to B_r. Thus, condition (1) holds.

Condition (2): Since $\mathcal{I}_t$ is an interpolant for $(\bigvee_{i=0}^{t} A_i)$ with respect to B_r, we know that $\mathcal{I}_t \wedge B_r$ is unsatisfiable. According to Algorithm 1, $B_0 = (\neg\kappa \wedge \neg Q)$ is one of the disjuncts of B_r. By Lemma 1, we know that $\mathcal{I}_t \wedge (\neg\kappa \wedge \neg Q)$ is also unsatisfiable. So, its negation, $\neg\mathcal{I}_t \vee \kappa \vee Q$, is a tautology. In addition, $(\neg\mathcal{I}_t \vee \kappa) \vee Q$ is equivalent to $\mathcal{I}_t \wedge \neg\kappa \implies Q$. Thus, condition (2) holds.

We also want to prove that if Algorithm 1 returns "no", the loop violates its postcondition Q. According to Algorithm 1, it returns "no" only when $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ is satisfiable and A_t is concrete. Since A_t is concrete, we can conclude that $A_0, A_1, \ldots, A_t$ are all concrete, and $(\bigvee_{i=0}^{t} A_i)$ represents the set of concrete states which the loop can reach from P within t iterations. In addition, B_r is the predicate representing the set of states from which the loop terminates within r iterations and violates the postcondition Q. Thus, a solution to the satisfiability problem of $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ is a counterexample starting from P and violating Q within $t + r$ iterations. ■

Discussion. In our implementation of Algorithm 1, we use an SMT solver to solve the formula $(\bigvee_{i=0}^{t} A_i) \wedge B_r$ in each iteration and directly obtain an interpolant from the solver if the formula is not satisfiable. However, the interpolant for an unsatisfiable formula is not unique, i.e., we may have many candidates fulfilling the three conditions in Definition 2. Which one is better is worthy of further investigation, and of course we need to define what "better" means first. Currently, our implementation relies on the quality of the interpolants returned by the SMT solver, which dominates the performance of our approach. In fact, initially, we have tried the Z3 [3] SMT solver. However, Z3 always returns the trivial interpolant $(\bigvee_{i=0}^{t} A_i)$, which makes our approach degenerate into traditi-

onal forward/backward analysis. Thus, we have tried another SMT solver, MathSAT [1], and it returns better interpolants than trivial ones, which makes our implementation effective to find inductive invariants (c.f. Section V). Techniques providing quality interpolants, like beautiful interpolants [6], could help to further improve the performance of our approach.

B. Backward Interpolation

In the previous approach, we obtain interpolants from forward reachability. In this section, we introduce another approach in the reverse direction. The basic idea is based on the following observation. Given an annotated loop, $\{P\}$ `while` κ `do` S `done` $\{Q\}$, if it does have an inductive invariant $\mathcal{I}$, then the negation of the invariant, $\neg\mathcal{I}$, must be inductive as well. That is, starting from $\neg\mathcal{I}$, no matter how many iterations we take backward for the loop, we will still be in the set of states satisfying $\neg\mathcal{I}$, i.e., $\mathsf{wp}(\mathsf{S}, \neg\mathcal{I}) \implies \neg\mathcal{I}$. If it were not the case, $\neg\mathcal{I}$ could cross over to $\mathcal{I}$ after some backward iterations so that $\mathcal{I}$ would not be inductive. This observation provides us another way to squeeze an inductive invariant. The intuition is that when we do the interpolation between the forward/backward reachabilities, i.e., the unsatisfiable formula $(\bigvee_{i=0}^{t} A_i) \wedge B_r$, we obtain an interpolant $\mathcal{I}$ for B_r instead of for $(\bigvee_{i=0}^{t} A_i)$. Then, we check whether the negation of the interpolant, $\neg\mathcal{I}$, is inductive or not. If yes, then $\neg\mathcal{I}$ is an inductive invariant, and we are done. Otherwise, we increase the values of t and r by one, respectively, and repeat the same process until we find an inductive invariant. Algorithm 2 shows the pseudo-code of the backward approach. Since Algorithm 2 is very similar to Algorithm 1 by symmetry, we omit its detailed descriptions here.

The soundness of Algorithm 2 is proved by Lemma 1 and Theorem 2. Notice that Algorithm 2 does not guarantee its termination. In practice, we can set an upper bound on the number of forward or backward iterations.

Theorem 2: Algorithm 2 is correct.

Proof 2: We want to prove that $\neg\mathcal{I}_r$ returned by Algorithm 2 satisfies the following three conditions: (1) $P \implies \neg\mathcal{I}_r$, (2) $\neg\mathcal{I}_r \wedge \neg\kappa \implies Q$, and (3) $\{\neg\mathcal{I}_r \wedge \kappa\}\ \mathsf{S}\ \{\neg\mathcal{I}_r\}$. Since $\neg\mathcal{I}_r$ is returned by Algorithm 2 only if condition (3) holds, we only have to prove conditions (1) and (2).

Condition (1): Since $\mathcal{I}_r$ is an interpolant for B_r with respect to $(\bigvee_{i=0}^{t} A_i)$, we know that $\mathcal{I}_r \wedge (\bigvee_{i=0}^{t} A_i)$ is unsatisfiable. By Lemma 1, $\mathcal{I}_r \wedge P$ is also unsatisfiable because $A_0 = P$ is one of the disjuncts of $(\bigvee_{i=0}^{t} A_i)$. Thus, $\neg P \vee \neg\mathcal{I}_r$ is a tautology. In addition, $\neg P \vee \neg\mathcal{I}_r$ is equivalent to $P \implies \neg\mathcal{I}_r$. Therefore, condition (1) holds.

Condition (2): Since $\mathcal{I}_r$ is an interpolant for B_r with respect to $(\bigvee_{i=0}^{t} A_i)$, we know $B_r \implies \mathcal{I}_r$. Let us consider the sequence of $B_0, B_1, \ldots, B_r$. Since $B_{j+1} = B_j \vee (\kappa \wedge \mathsf{wp}(\mathsf{S}, B_j))$ for $0 \leq j < r$, we know $B_0 \implies B_1 \implies \ldots \implies B_r$. Thus, $B_0 = \neg\kappa \wedge \neg Q \implies \mathcal{I}_r$, which is logically equivalent to $(\kappa \vee Q) \vee \mathcal{I}_r$. According to the associativity law, $(\kappa \vee Q) \vee \mathcal{I}_r$ is equivalent to $(\kappa \vee \mathcal{I}_r) \vee Q$, which is also logically equivalent to $\neg\kappa \wedge \neg\mathcal{I} \implies Q$. Therefore, condition (2) holds. ■

Algorithm 2: Squeeze Invariant – Backward

input : An annotated loop: $\{P\}$ `while` κ `do` S `done` $\{Q\}$
output: (yes/no, $\mathcal{I}$), where $\mathcal{I}$ is a loop invariant

```
1  A_0 ⟵ P ;
2  B_0 ⟵ (¬κ ∧ ¬Q) ;
3  t ⟵ r ⟵ 0 ;
4  while True do
5      if B_r ∧ (⋁_{i=0}^{t} A_i) is not satisfiable then
6          Let I_r be the interpolant for B_r w.r.t. (⋁_{i=0}^{t} A_i) ;
7          if {¬I_r ∧ κ} S {¬I_r} then return (yes, ¬I_r) ;
8          B_{r+1} ⟵ I_r ∨ (κ ∧ wp(S, I_r)) ;
9          r ⟵ r + 1 ;
10         A_{t+1} ⟵ sp(A_t ∧ κ, S) ;
11         t ⟵ t + 1 ;
12     else
13         if B_r is concrete then return (no, ⊥) ;
14         else
15             while B_r is not concrete do r ⟵ r − 1 ;
16             B_{r+1} ⟵ B_0 ∨ (κ ∧ wp(S, B_r)) ;
17             r ⟵ r + 1 ;
```

C. Extensions

Our approaches have been proposed to handle one single-level loop. In this section, we discuss how to extend our approaches to handle general loop structures, e.g., nested loops or a sequence of loops. Fig. 8 (a) shows the general structure of a two-level nested loop, where we only have the precondition and postcondition of the outer loop. In this case, we are not able to perform our approaches on the inner loop because we do not know its precondition and postcondition. To handle this case, one straightforward solution is to flatten the nested loops into one single-level loop. Fig. 8 (b) illustrates how the flattening can be done. Briefly, we just introduce an auxiliary variable, `block`, indicating which loop is active now. One can easily verify that the flattened loop is equivalent to the original one. If the outer loop has more than one loop inside, we can inductively perform the flattening. Once the loop is flattened, our approaches apply.

The second case that our approaches are not directly applicable is a sequence of loops. Fig. 9 (a) shows the general structure of two loops in a sequence. We can see that the postcondition of the first loop as well as the precondition of the second are missing. To bridge the gap, we can combine them into one single-level loop. Fig. 9 (b) illustrates how the translation is done. Similarly, we use auxiliary variable, `block`, to indicate the active loop. One can easily verify that the translation is correct. If we have more than two loops in a sequence, we can inductively perform the translation such that our approaches apply. In a general program, we may have nested loops and sequence loops mixed together. In such a situation, we can inductively perform the translation based on

```
while(κ1){
    S1;

    while(κ2){
        S2;
    }

    S1';
}
```

(a) Nested Loops

```
int block = 0;

while (block != -1){
   if (block == 0){
      if (κ1) {
         S1; block = 1;
      } else {
         block = -1;
      }
   } else if(block == 1){
      if (κ2){ S2; }
      else { S1'; block = 0; }
   }
}
```

(b) Flattened Loop

Fig. 8. Flattening Nested Loops into a Flattened Loop

```
while(κ1){
    S1;
}

S;

while(κ2){
    S2;
}
```

(a) Sequential Loops

```
int block = 0;

while (block != -1){
   if (block == 0){
      if (κ1){ S1; }
      else { S; block = 1; }
   } else if (block == 1) {
      if (κ2) { S2; }
      else { block = -1; }
   }
}
```

(b) Combined Loop

Fig. 9. Translation of a Sequence of Loops into One Loop

the two primitive cases in Fig. 8 and Fig. 9.

One more interesting extension is to combine the forward and backward approaches into a bidirectional one. That is, in each forward/backward advancing iteration, we obtain the forward interpolant as well as the backward interpolant simultaneously. If either of them is inductive, we are done; otherwise, we start from them to advance in the next iteration. If they intersect with each other, we backtrack to the concrete cases and then continue from there. The process is repeated until an invariant is found.

V. Evaluation

The proposed approach to automatically generating inductive loop invariants has been implemented in a tool, called F*i*B. In the implementation, we use the SMT solver, MathSAT [1], to solve the satisfiability problems as well as to obtain interpolants if the formula is not satisfiable.

To evaluate our approach, we compared F*i*B with six existing state-of-the-art invariant generation tools: BLAST [33], InvGen [31], Interproc [35], CPAchecker [9], ITP [41], and HOLA[20]. Each of the compared tools represents a different family of invariant generation techniques. BLAST is a CEGAR-based model checker that generates loop invariants from counterexamples based on interpolations. InvGen assumes that the invariant is of a given template form and generates loop invariants by solving constraints with unknown parameters. Interproc generates loop invariants based on abstract interpretation. CPAchecker is a configurable software

```
1  int x = 0;
2  int y = 0;
3
4  while(x != 100){
5     x = x + 1;
6     y = y + 1;
7  }
8
9  assert(y == 100);
```

I : $x = 0 \wedge y = 0 \wedge pc = 4$

t_1 : $(pc = 4) \wedge (x \neq 100) \wedge (pc' = 5) \wedge (x' = x) \wedge (y' = y)$
t_2 : $(pc = 4) \wedge (x = 100) \wedge (pc' = 9) \wedge (x' = x) \wedge (y' = y)$
t_3 : $(pc = 5) \wedge (pc' = 6) \wedge (x' = x + 1) \wedge (y' = y)$
t_4 : $(pc = 6) \wedge (pc' = 4) \wedge (y' = y + 1) \wedge (x' = x)$
t_5 : $(pc = 9) \wedge (pc' = 9) \wedge (x' = x) \wedge (y' = y)$

T : $t_1 \vee t_2 \vee t_3 \vee t_4 \vee t_5$

$\neg\varphi$: $pc = 9 \wedge y \neq 100$

Fig. 10. Transition Relation Construction

model checker which supports CEGAR based verification approaches. HOLA generates loop invariants based on logical abduction and quantifier elimination. ITP is a transition-relation-based model checking technique, which calculates fixpoints by interpolation. For our experiments on ITP, we adopt the standard approach (Chapter 2 of [14]) to encode the transition relation of a program. Fig. 10 shows an example illustrating the encoding. An auxiliary variable pc is required to indicate the program counter (line number). For example, if the next statement to be executed is `x = x + 1`, that is, $pc = 5$, then the statement is encoded as t_3 in Fig. 10. The overall transition relatioin would be $T : t_1 \vee t_2 \vee t_3 \vee t_4 \vee t_5$. The initial condition would be $I : x = 0 \wedge y = 0 \wedge pc = 4$, and the assertion checking problem becomes the reachability problem of the formula: $pc = 9 \wedge y \neq 100$. We have implemented the ITP approach in our F*i*B tool for exeperiments. Both ITP and our approach use MathSAT as their satisfiable checking and interpolation engine with the same configuration.

HOLA was already compared with BLAST, InvGen, and Interproc in [20] through a set of benchmarks collected from other loop invariant generation papers [10], [29], [8], [36], [27], InvGen test suite [5], NECLA verification benchmarks [4], and the HOLA test suite [20]. Considering that the set of benchmarks is rather comprehensive, we adopted the same set for evaluation. All the details of benchmarks and tools are collected and can be found in [2]. Out of the original 46 benchmarks, we filtered out those with assertions (postconditions) inside the loops and selected the 41 benchmarks for evaluation. Each benchmark has at least one loop and at least one assertion, and some benchmarks have nested loops or sequence of loops. If the benchmark has nested or sequence loops, it is manually translated into a single loop using the translation mentioned in Section IV-C. Table I shows the experimental results that are obtained on a machine, with an Intel Xeon 3.5GHz CPU and 16GB memory, running the 64-bit Ubuntu version 16.04.

TABLE I
EXPERIMENTAL RESULTS

No.	LoC	BLAST		InvGen		Interproc		CPAchecker		ITP		HOLA		F*i*B-F		F*i*B-B		F*i*B-Bi	
			Time (sec)		Time (sec)		Time (sec)		Time (sec)		Time (sec)		Time (sec)		Time (sec)		Time (sec)		Time (sec)
1	21	✔	0.28	✔	0.27	✔	0.04	✔	0.79	✔	1.84	✔	0.04	✔	0.48	✔	0.05	✔	0.11
2	26	✗	0.13	✗	0.17	✗	0.10	✔	1.21	✔	14.05	✔	0.31	✔	0.32	✔	0.19	✔	0.26
4	21	✔	3.72	✗	0.23	✗	0.03	✔	0.74	✔	0.21	✔	0.02	✔	0.01	✔	0.01	✔	0.01
5	27		◷	✔	0.31	✔	0.13		◷	✔	16.54	✔	0.04	✔	0.17	✔	0.04	✔	0.95
6	28	✗	0.14	✗	0.15	✔	0.09		◷		◷	✔	0.20	✔	0.18	✔	0.26	✔	0.31
7	27	✔	11.5	✔	0.56	✔	0.08	✔	1.07	✔	16.22	✔	0.62	✔	0.01	✔	0.02	✔	0.03
8	30	✔	2.61	✔	0.44	✗	0.07	✔	1.16		◷	✔	0.38	✔	0.01	✔	0.01	✔	0.01
9	49		◷	✗	0.17	✔	0.08		◷	✔	0.20	✔	0.08	✔	0.03	✔	0.02	✔	0.02
10	30	✔	1.27	✗	0.20	✗	0.07	✔	0.86	✔	4.53	✔	0.13	✔	0.03	✔	0.01	✔	0.02
11	24		◷	✔	0.36	✔	0.02		◷	✔	0.35	✔	0.13	✔	0.01	✔	0.01	✔	0.01
12	34		◷	✗	0.25	✗	0.12		◷		◷	✔	3.13	✔	1.93	✔	1.69	✔	2.01
13	25	✔	0.76	✗	0.44	✔	0.04	✔	0.87	✔	0.94	✔	0.38	✔	0.01	✔	0.01	✔	0.03
14	26	✔	5.30	✔	0.58	✔	0.03	✔	1.10	✔	5.67	✔	1.07	✔	0.01	✔	0.01	✔	0.03
15	28	✔	0.78	✔	0.28	✔	0.02	✔	0.91	✔	0.63	✗	11.7	✔	0.01	✔	0.01	✔	0.01
16	23	✗	1.69	✔	0.38	✔	0.02	✔	0.89	✔	0.03	✔	0.12	✔	0.01	✔	0.01	✔	0.01
17	22	✔	1.14	✗	0.20	✔	0.08	✔	0.89		◷	✔	0.09	✔	0.15	✔	0.33	✔	0.19
18	24		◷	✔	7.26	✗	0.03		◷	✔	0.60	✔	2.75	✔	0.02	✔	0.01	✔	0.02
19	24	✔	3.74	✗	0.39	✗	0.03	✔	1.31	✔	30.09	✗	14.7	✔	0.04	✔	0.02	✔	0.04
20	33	✔	5.02	✔	0.74	✗	0.07	✔	1.05	✔	5.73	✔	2.56	✔	0.02	✔	0.02	✔	0.05
21	39	✗	1.84	✗	0.14	✔	0.09	✔	0.96		◷	✔	0.59		◷	✔	0.13		◷
22	26	✗	0.15	✗	0.16	✗	0.03	✔	1.20	✔	21.88	✔	0.26	✔	0.14	✔	0.14	✔	0.18
23	20		◷	✔	0.38	✗	0.05		◷	✔	1.60	✔	0.05	✔	1.74	✔	0.02	✔	0.01
25	33	✔	1.33	✗	0.20	✗	0.05	✔	0.86		◷	✔	0.04	✔	0.09	✔	0.05	✔	0.06
26	24	✗	0.12	✗	0.18	✗	0.09		◷		◷	✔	0.31	✔	0.80	✔	0.49	✔	0.81
28	25	✗	0.94	✔	0.43	✔	0.02		◷	✔	0.26	✔	0.06	✔	0.01	✔	0.01	✔	0.01
29	32	✗	0.09	✗	0.17	✗	0.14	✔	0.86	✔	7.41	✔	0.31	✔	0.05	✔	0.02	✔	0.02
30	22	✔	0.70	✔	0.40	✗	0.01		◷	✔	2.92	✔	0.03	✔	0.32	✔	0.01	✔	0.16
32	24		◷	✗	0.18	✗	0.05		◷	✔	169.97	✔	0.65	✔	0.13	✔	0.08	✔	0.18
33	36	✗	1.72	✗	0.17	✗	0.12	✔	1.12	✔	48.32	✔	0.09	✔	0.09	✔	0.05	✔	0.10
34	23	✗	4.12	✗	0.19	✗	0.03	✔	1.75		◷		◷	✔	0.73	✔	0.33	✔	0.91
35	17	✔	0.39	✗	0.43	✗	0.02	✔	0.90	✔	0.06	✔	0.12	✔	0.01	✔	0.01	✔	0.01
36	71	✗	3.21	✗	0.24	✗	0.60	✔	54.31		◷	✔	0.89		◷	✔	0.23	✔	0.46
37	21	✔	2.01	✔	0.42	✗	0.03	✔	0.80	✔	1.47	✔	0.41	✔	0.01	✔	0.01	✔	0.01
38	20	✗	0.29	✗	0.21	✗	0.05	✔	0.99	✔	11.78	✔	0.27	✔	0.02	✔	0.04	✔	0.05
40	30	✗	3.97	✗	0.17	✗	0.11	✔	5.76		◷	✔	0.80	✔	0.04	✔	0.06	✔	0.06
41	25	✗	0.01	✔	0.38	✗	0.04	✗	1.03	✔	3.84	✔	0.43	✔	0.21	✔	0.03	✔	0.15
42	37	✗	0.06	✗	0.15	✔	0.06	✔	65.52	✔	16.37	✔	0.51	✔	0.06	✔	0.10	✔	0.08
43	27	✔	0.17	✗	0.14	✔	0.05	✔	0.72	✔	0.07	✔	0.07	✔	0.01	✔	0.01	✔	0.01
44	35	✔	1.33	✗	0.44	✗	0.05	✔	0.88	✔	0.48	✔	1.27	✔	0.04	✔	0.01	✔	0.01
45	44	✗	0.55	✗	0.20	✗	0.58		◷		◷	✔	0.60	✔	0.06	✔	0.11	✔	0.08
46	24	✗	0.23	✗	0.21	✗	0.04	✔	0.88	✔	10.62	✔	0.19	✔	0.02	✔	0.02	✔	0.04

Each benchmark number in the first column of Table I corresponds to that in [20], and all the assertions in benchmarks are satisfied. For each tool, the symbol ✔ indicates that the tool is able to verify all assertions in the benchmark, while the symbol ✗ indicates that the tool is not able to infer the loop invariants to verify the assertions, or the verification result is wrong. The columns labeled "Time" indicate the execution time (in seconds) to generate loop invariants for verifying the assertions. The symbol ◷ indicates that the tool did not terminate in 200 seconds. We mark the least execution time of correct verification in gray color for each benchmark.

As Table I shows, the proposed approaches are very effective to construct loop invariants to prove the assertions within 2 seconds for each benchmark, especially the backward interpolation approach (F*i*B-B). The forward interpolation approach (F*i*B-F) and the bidirectional approach (F*i*B-Bi) failed to prove the assertion in benchmark 21 within 200 seconds. Among the other six tools, HOLA performs the best, which only failed to verify three benchmarks (wrong verification results in no. 15 and no. 19, and timeout in no. 34). Generally, we found that the backward interpolation approach (F*i*B-B) outperforms the forward approach (F*i*B-F) and bidirectional approach (F*i*B-Bi), especially for benchmarks 5, 21, 23, and 41.

Table II shows the statistics of our three approaches including the number of forward iterations (f), the number of forward backtracks (fb), the number of backward iterations (b), the number of backward backtracks (bb), whether the invariant is obtained based on abstract interpolants or concrete states (a/c), and whether the obtained invariant is disjunctive ($\vee$?). The size of the obtained invariant refers to the number of nodes in its syntax tree, e.g. the size of $(x = y)$ is three. If the benchmark has more than one assertion to be checked, we list the average size of the invariant for all the assertions. We also show the size of the fixpoints found by the ITP approach. Since ITP aims to find a fixpoint instead of an inductive invariant, we

TABLE II
EXPERIMENT STATISTICS

No.	ITP			F*i*B-F						F*i*B-B						F*i*B-Bi						
	abort	k	size	f	fb	b	a/c	size	∨?	f	b	bb	a/c	size	∨?	f	fb	b	bb	a/c	size	∨?
1	2	9	3,842	2	1	1	c	2	no	1	1	0	a	13	yes	2	1	2	1	c	18	no
2	3	18	2,914	4	2	2	c	135	no	2	2	0	a	193	yes	4	2	4	2	c	150	no
4	1	8	279	1	0	1	a	20	yes	0	0	0	c	7	yes	0	0	0	0	c	7	yes
5	4	39	18,914	12	6	6	c	82	yes	2	2	0	a	43	yes	22	11	22	11	c	450	no
6	–	–	–	2	1	1	c	85	yes	1	1	0	a	62	yes	2	1	2	1	c	62	yes
7	4	28	25,522	2	1	1	c	30	yes	1	1	0	a	25	no	2	1	2	1	c	63	no
8	–	–	–	0	0	0	c	13	yes	0	0	0	c	8	yes	0	0	0	0	c	8	yes
9	0	4	140	0	0	0	c	3	no	0	0	0	c	4	no	0	0	0	0	c	4	no
10	3	18	13,454	2	1	1	c	16	yes	1	1	0	a	22	no	2	1	2	1	c	22	no
11	2	9	426	2	1	1	c	30	yes	1	1	0	a	20	no	2	1	2	1	c	23	no
12	–	–	–	8	4	4	c	529	yes	4	4	0	a	408	yes	8	4	8	4	c	221	no
13	3	13	1,982	2	1	1	c	27	yes	1	1	0	a	25	yes	2	1	2	1	c	33	no
14	3	24	5,703	4	2	2	c	35	yes	1	1	0	a	18	no	4	2	4	2	c	20	no
15	2	12	747	6	3	3	c	16	yes	2	2	0	a	16	no	4	2	4	2	c	18	no
16	0	4	105	0	0	0	c	18	yes	0	0	0	c	20	yes	0	0	0	0	c	20	yes
17	–	–	–	6	3	3	c	115	yes	9	10	1	a	475	yes	6	3	6	3	c	115	no
18	2	12	770	2	1	1	c	28	yes	1	1	0	a	30	no	2	1	2	1	c	28	no
19	4	48	37,318	4	2	2	c	29	yes	2	2	0	a	56	yes	4	2	4	2	c	64	yes
20	2	14	4,225	0	0	0	c	10	yes	0	0	0	c	10	yes	0	0	0	0	c	10	yes
21	–	–	–	–	–	–	–	–	–	2	2	0	a	53	yes	–	–	–	–	–	–	–
22	2	17	1,445	4	2	2	c	58	yes	2	2	0	a	38	yes	4	2	4	2	c	43	yes
23	3	19	2,911	77	37	40	c	345	no	1	1	0	a	11	no	6	3	6	3	c	36	yes
25	–	–	–	2	1	1	c	29	yes	1	1	0	a	14	no	2	1	2	1	c	29	yes
26	–	–	–	2	1	1	c	157	no	1	1	0	a	209	yes	2	1	2	1	c	143	no
28	2	9	266	0	0	0	c	3	no	0	0	0	c	3	no	0	0	0	0	c	3	no
29	2	10	18,095	0	0	0	c	14	no	0	0	0	c	15	no	0	0	0	0	c	15	no
30	4	35	1,775	32	16	16	c	208	no	1	1	0	a	11	no	22	11	22	11	c	28	no
32	4	42	27,060	6	3	3	c	104	yes	3	3	0	a	126	no	6	3	6	3	c	81	no
33	2	18	138,121	2	1	1	c	69	yes	1	1	0	a	39	no	2	1	2	1	c	79	no
34	–	–	–	20	7	13	a	406	no	16	20	4	a	242	yes	28	11	28	11	c	182	no
35	0	4	104	0	0	0	c	8	yes	0	0	0	c	8	no	0	0	0	0	c	8	no
36	–	–	–	–	–	–	–	–	–	1	1	0	a	85	no	2	1	2	1	c	85	no
37	2	15	3,673	0	0	0	c	14	yes	0	0	0	c	20	yes	0	0	0	0	c	20	yes
38	3	27	13,414	2	1	1	c	40	yes	1	1	0	a	45	no	2	1	2	1	c	42	no
40	–	–	–	0	0	0	c	17	yes	0	0	0	c	16	yes	0	0	0	0	c	16	yes
41	5	34	3,257	26	12	14	c	508	yes	1	1	0	a	12	no	16	8	16	8	c	236	yes
42	3	18	29,392	2	1	1	c	74	yes	1	1	0	a	55	no	2	1	2	1	c	72	yes
43	0	4	184	0	0	0	c	8	yes	0	0	0	c	8	no	0	0	0	0	c	8	no
44	2	11	747	8	0	8	a	34	yes	0	0	0	c	9	yes	0	0	0	0	c	9	yes
45	–	–	–	0	0	0	c	17	yes	0	0	0	c	13	yes	0	0	0	0	c	13	yes
46	2	21	20,207	2	1	1	c	20	yes	1	1	0	a	18	no	2	1	2	1	c	18	no

can observe that the size of a fixpoint is usually much larger than that of an inductive invariant. More discussions about why our approach outperforms ITP can be found in Section VI.

We can also observe that the backward (F*i*B-B) approach has zero backtracks except benchmarks no. 17 and 34, which explains why it performs the best among the proposed three approaches. After our investigation based on the statistics given in Table II, we summarize the reasons why F*i*B-B is the best: (1) The assertion (postcondition) to be proved is usually more symbolic (abstract), e.g., $x > 0$, than the precondition of the loop (e.g., $x = 0 \wedge y = 0$). Thus, the interpolant for the backward reachability would converge to be inductive more easily. (2) For the assignment statement, the weakest precondition (wp) calculation is faster than the strongest postcondition (sp) calculation because wp does not require quantifier elimination (except the nondeterminism assignment statement), but sp does. (3) Considering the assertion of a loop is usually more symbolic than its precondition, after the same number of iterations, the backward analysis based on wp provides more general information than forward analysis based on sp. Thus, the interpolant for the backward direction would be more close to an inductive invariant, which can be confirmed in the a/c column in Table II by the fact that F*i*B-B often obtains an inductive invariant based on abstract interpolants from previous iterations. That is why F*i*B-B has much less backward backtracks than F*i*B-F.

VI. RELATED WORKS AND DISCUSSIONS

Automatic loop invariant generation is a fundamental problem in program analysis and verification. Theoretically, it is an undecidable problem. To tackle this problem in practice, both static and dynamic analysis based techniques have been proposed. Static analysis based techniques can be further classified to the following categories based on the underlying techniques that are used: abstract interpretation [17], [44], [16], [37], [38], SMT solving based techniques such as counterexample guided abstraction refinement (CEGAR) [33], [7], [13], constraint-based methods [31], [15], [30], Craig interpo-

lation [43], [32], [41], [42], [40], abduction [45], [11], [26], [28], [12], [20], learning-based approach [39], and algebraic approaches [47], [46], [48]. Dynamic analysis based techniques include guess-and-check techniques [22], [21], [49], [50], [51] and learning-based techniques [24], [25]. Our approach belongs to the static analysis category.

The very related work to ours is ITP [41], in which a transition system (I, T), where I is the initial condition and T is the transition relation, as well as a safety property φ are considered. Given a value k, their approach constructs a BMC formula in $k+1$ steps: $I \wedge T \wedge T^k \wedge \neg\varphi$, and obtains an interpolant $\mathcal{I}$ for $I \wedge T$ with respect to $T^k \wedge \neg\varphi$. That is, $\mathcal{I}$ is an over-approximation of states that can be reachable within one step and not going to violate φ within k steps. Then, $\mathcal{I}$ is considered in the formula: $\mathcal{I} \wedge T \wedge T^k \wedge \neg\varphi$. If it is not satisfiable, an interpolant $\mathcal{I}'$ for $\mathcal{I} \wedge T$ with respect to $T^k \wedge \neg\varphi$ is an over-approximation of state that can be reachable in two steps and not going to violate φ within k steps. However, if the formula is satisfiable, then nothing can be concluded, and the algorithm aborts. The value of k needs to be increased by a certain value, and the same process is repeated until a real counterexample is found or the obtained interpolant reaches a fix-point. If ITP advances t steps and then aborts, we increase k by t in the next iteration. Table II shows the number of aborts and the sufficient value of k to have a conclusive result. In our experiments, we start with $k = 4$, which is small but sufficient to find fix-points, e.g., benchmark no. 9, 16, 35, and 43.

Our approach is different from ITP in several aspects. We list the differences and try to analyze why our approach outperforms ITP in the followings:

- Our approach obtains the concrete forward/backward reachability and tries to squeeze out an inductive invariant based on the interpolation in between; ITP over-approximates the set of reachable states for each step with a k-lookhead to see if a fixpoint can be quickly reached based on the abstraction. If our approach fails to find an inductive invariant in one iteration, we only have to advance one further step from the current reachability for the next iteration because the reachability is concrete and can be accumulated. However, if ITP fails to find a fixpoint in one iteration, it has to be restarted from scratch with a new k larger than the previous one, whose value is tricky to decide because if the increment of k is too small, many restarts may be required; if the increment is too big, the satisfiability checking problem may become difficult to solve.
- Our approach handles program statements compositionally. As we know, quantifier elimination is computationally expensive [41]. However, in our approach, we process one program statement at a time; thus, only one quantified variable has to be eliminated at a time. In addition, only assignment statements require quantifier elimination. Thus, quantifier elimination is not a performance bottleneck in our approach, which is also confirmed in our experiments. On the contrary, the ITP approach considers the global transition relation of all program statements. If there are n variables in the program, the k-step BMC problem consists of $k \times n$ variables. If the value of k is large, solving the BMC formula takes longer time, as evidenced by benchmark no. 19 and 32. Furthermore, if the transition relation of a program is lengthy, the BMC problem also becomes difficult to solve.

Another closely related work is DAR [52], which also considers a transition system (I, T) and a safety property φ. Their approach obtains two interpolation sequences: (1) The *forward interpolation sequence* $\langle (F_0 = I), F_1, F_2, \ldots, F_k \rangle$ satisfying $F_i \wedge T \implies F_{i+1}$ for $0 \leq i < k$ and $F_i \implies \varphi$ for $0 \leq i \leq k$. (2) The *backward interpolation sequence* $\langle (B_0 = \neg\varphi), B_1, B_2, \ldots, B_k \rangle$ satisfying $B_i \wedge T \implies B_{i+1}$ for $0 < i \leq k$ and $B_i \implies \neg I$ for $0 \leq i \leq k$. The two interpolation sequences are strengthened or extended by the local and global strengthening procedures until a counterexample is found or either interpolation sequence reaches a fix-point.

Our approach is different from DAR in several aspects. We list the differences as follows:

- Our approach handles program statements compositionally, while the DAR approach, similar to ITP, considers the global transition relation.
- What DAR maintains are sequences of abstractions (interpolants). Once each interpolation sequence is changed or extended, some subsequent process has to be performed so that the properties mentioned above are still valid. What our approach maintains are concrete forward and backward reachable states step by step, whose validity is not changed. They can be accumulated for the following iterations without any recalculations.

We have tried to obtain the DAR tool on internet for evaluation, but failed. According to the experimental results reported in [52], the performance of DAR and ITP are evenly balanced. We are interested in implementing the DAR approach by ourselves in the future for evaluation.

VII. Conclusion and Future Work

In this work, we propose an novel approach to automatically constructing inductive loop invariants, which solves the fundamental problem in program analysis and verification. Our approach squeezes an inductive invariant based on Craig interpolants between forward and backward predicate transformers. In the future, we would like to investigate the quality of interpolants.

Acknowledgment

This research is mainly supported by the startup grant M4081588.020.500000 of School of Computer Science and Engineering in Nanyang Technological University and partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] http://mathsat.fbk.eu/.
[2] https://github.com/spencerxiao/ase2017-results-and-tools.
[3] https://z3.codeplex.com/.
[4] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php.
[5] http://www.tcs.tifr.res.in/ãgupta/invgen/.
[6] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, volume 8044 of *LNCS*, pages 313–329, 2013.
[7] T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264, 2001.
[8] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, volume 42, pages 300–309, 2007.
[9] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
[10] A. Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing*, volume 7317 of *LNCS*, pages 1–14, 2012.
[11] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
[12] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *Programming Languages and Systems (APLAS)*, volume 5904 of *LNCS*, pages 259–274, 2009.
[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
[15] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–432, 2003.
[16] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
[17] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
[18] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.
[19] E. W. Dijkstra and C. S. Scholten. Predicate calculus and program semantics. *Texts and Monographs in Computer Science*, 1990.
[20] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, volume 48, pages 443–456, 2013.
[21] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.
[22] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, 2001.
[23] R. W. Floyd. Assigning meanings to programs. In *Applied Math*, volume XIX, pages 19–32, 1967.
[24] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.
[25] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, volume 51, pages 499–512, 2016.
[26] R. Giacobazzi. Abductive analysis of modular logic programs. In *International Symposium on Logic Programming*, pages 377–391, 1994.
[27] B. Gulavani and S. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, pages 474–488, 2006.
[28] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
[29] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, volume 43, pages 281–292, 2008.
[30] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 120–135, 2009.
[31] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640, 2009.
[32] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
[33] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *International Conference on Model Checking Software (SPIN)*, volume 5403 of *LNCS*, pages 235–239, 2003.
[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
[35] B. Jeannet. Interproc analyzer for recursive programs with numerical variables.
[36] R. Jhala and K. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
[37] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
[38] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 229–244, 2009.
[39] W. Lee, Y. Jung, B.-Y. Wang, and K. Yi. Predicate generation for learning-based quantifier-free loop invariant inference. *Logical Methods in Computer Science*, 8(3):1–21, 2012.
[40] S.-W. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong. Interpolation guided compositional verification. In *ASE*, pages 65–74, 2015.
[41] K. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13, 2003.
[42] K. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136, 2006.
[43] K. McMillan. Lazy annotation for program testing and verification. In *CAV*, volume 6174 of *LNCS*, pages 104–118, 2010.
[44] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
[45] C. Peirce. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.
[46] R. Rebiha, A. V. Moura, and N. Matringe. Generating invariants for non-linear loops by linear algebraic methods. *Formal Aspects of Computing*, 27(5):805–829, 2015.
[47] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.
[48] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.
[49] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, volume 8559 of *LNCS*, pages 88–105, 2014.
[50] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems*, volume 7792 of *LNCS*, pages 574–592, 2013.
[51] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis*, volume 7935 of *LNCS*, pages 388–411, 2013.
[52] Y. Vizel, O. Grumberg, and S. Shoham. Intertwined forward-backward reachability analysis using interpolants. In *TACAS*, pages 308–323, 2013.
[53] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.