SymInfer: Inferring Program Invariants using Symbolic States

ThanhVu Nguyen
University of Nebraska-Lincoln, USA
tnguyen@cse.unl.edu

Matthew B. Dwyer
University of Nebraska-Lincoln, USA
dwyer@cse.unl.edu

Willem Visser
Stellenbosch University, South Africa
wvisser@cs.sun.ac.za

***Abstract*—We introduce a new technique for inferring program invariants that uses symbolic states generated by symbolic execution. Symbolic states, which consist of path conditions and constraints on local variables, are a compact description of sets of concrete program states and they can be used for both invariant inference and invariant verification. Our technique uses a counterexample-based algorithm that creates concrete states from symbolic states, infers candidate invariants from concrete states, and then verifies or refutes candidate invariants using symbolic states. The refutation case produces concrete counterexamples that prevent spurious results and allow the technique to obtain more precise invariants. This process stops when the algorithm reaches a stable set of invariants.**

We present SymInfer, a tool that implements these ideas to automatically generate invariants at arbitrary locations in a Java program. The tool obtains symbolic states from Symbolic PathFinder and uses existing algorithms to infer complex (potentially nonlinear) numerical invariants. Our preliminary results show that SymInfer is effective in using symbolic states to generate precise and useful invariants for proving program safety and analyzing program runtime complexity. We also show that SymInfer outperforms existing invariant generation systems.

I. Introduction

Program invariants describe properties that always hold at a program location. Examples of invariants include pre/post-conditions, loop invariants, and assertions. Invariants are useful in many programming tasks, including documentation, testing, debugging, verification, code generation, and synthesis [1]–[4].

Daikon [2] demonstrated that dynamic analysis is a practical approach to infer invariants from *concrete program states* that are observed when running the program on sample inputs. Dynamic inference is typically efficient and supports expressive invariants, but can often produce spurious invariants that do not hold for all possible inputs. Several invariant generation aproaches (e.g., iDiscovery [5], PIE [6], ICE [7], NumInv [8]) use a hybrid approach that dynamically infers candidate invariants and then statically checks that they hold for all inputs. For a spurious invariant, the checker produces counterexamples, which help the inference process avoid this invariant and obtain more accurate results. This approach, called *CounterExample Guided Invariant Generation* (CEGIR), iterates the inference and checking processes until achieving stable results.

In this paper, we present a CEGIR technique that uses *symbolic program states*. Our key insight is that symbolic states generated by a symbolic execution engine are (a) compact encodings of large (potentially infinite) sets of concrete states, (b) naturally diverse since they arise along different execution paths, (c) explicit in encoding relationships between program variables, (d) amenable to direct manipulation and optimization, such as combining sets of states into a single joint encoding, and (e) reusable across many different reasoning tasks within CEGIR algorithms.

We define algorithms for symbolic CEGIR that can be instantiated using different symbolic execution engines and present an implementation SymInfer that uses Symbolic PathFinder [9] (SPF)— a symbolic executor for Java. SymInfer uses symbolic states in both the invariant inference and verification processes. For inference, SymInfer uses symbolic states to generate concrete states to bootstrap a set of candidate invariants using DIG [10]–[12]—which can infer expressive nonlinear invariants. For verification, SymInfer formulates verification conditions from symbolic states to confirm or refute an invariant, solves those using a SAT solver, and produces counterexamples to refine the inference process.

Symbolic states allow SymInfer to overcome several limitations of existing CEGIR approaches. iDiscovery, ICE, and PIE are limited to computing relatively simple invariants and often do not consider complex programs with nonlinear arithmetic and properties such as $x = qy + r, x^2 + y^2 = z^2$. These invariants appear in safety and security-critical software and can be leveraged to improve quality, e.g., to verify the absence of errors in Airbus avionic systems [13] and to analyze program runtime complexity to detect security threats [14], [15]. As our evaluation of SymInfer demonstrates in Sec. V, iDiscovery, which uses Daikon for inference, does not support nonlinear properties, and both ICE and PIE timeout frequently when nonlinear arithmetic is involved. Recent work on NumInv [8] also uses DIG to infer invariants, but it invokes KLEE [16] as a blackbox verifier for candidate invariants. Since KLEE is unaware of the goals of its verification it will attempt to explore the entire program state space and must recompute that state space for each candidate invariant. In contrast, SymInfer constructs a fragment of the state space that generates a set of symbolic states that is sufficiently diverse for invariant verification and it reuses symbolic states for all invariants.

We evaluated SymInfer over 3 distinct benchmarks which consist of 92 programs. The study shows that SymInfer: (1) can generate complex nonlinear invariants required in 21/27 NLA benchmarks, (2) is effective in finding nontrivial complexity bounds for 18/19 programs, with 4 of those improving

```
int idiv(int x1, int x2) {
  assert(x1 >= 0 && x2 >= 1);
  int y1,y2,y3;
  y1 = y2 = 0; y3 = x1;
  while (y3 != 0) //[L]
    if (y2 + 1 == x2) {
      y1 = y1 + 1;
      y2 = 0;
      y3 = y3 - 1;
    } else {
      y2 = y2 + 1;
      y3 = y3 - 1;
    }
  return y1;
}
```

concrete L-states:

$x1$	$x2$	$y1$	$y2$	$y3$
15	2	0	0	15
15	2	0	1	14
15	2	1	0	13
		⋮		
4	1	0	0	4
4	1	1	0	3
		⋮		

Fig. 1. An integer division program and concrete L-states observed on inputs $(x1 = 15,\ x2 = 2)$ and $(x1 = 4,\ x2 = 1)$

on the best known bounds from the literature, (3) improves on the state-of-the-art PIE tool in 41/46 programs in the HOLA benchmark, and (4) outperforms NumInv across the benchmarks while computing similar or better invariants.

These results strongly suggest that symbolic states form a powerful basis for computing program invariants. They permit an approach that blends the best features of dynamic inference techniques and purely symbolic techniques, such as weakest-precondition reasoning. The key contribution of our work lies in the identification of the value of symbolic states in CEGIR, in developing an algorithmic framework for adaptively computing a sufficient set of symbolic states for invariant inference, and in demonstrating, through our evaluation of SymInfer, that it improves on the best known techniques.

II. Overview

We illustrate invariant inference using symbolic states on the integer division algorithm in Figure 1; `L` marks the location at which we are interested in computing invariants. This example states assumptions on the values of the parameters, e.g., no division by zero. The best invariant at L is $\texttt{x2}\cdot\texttt{y1}+\texttt{y2}+\texttt{y3} = \texttt{x1}$. This loop invariant encodes the precise semantics of the loop computing integer division, i.e., the dividend `x1` equals the divisor `x2` times the quotient `y1` plus the remainder, which is the sum of the two temporary variables `y2` and `y3`.

Existing methods of dynamic invariant inference would instrument the program at location `L` to record values of the 5 local variables, and then, given a set of input vectors, execute the program to record a set of *concrete states* of the program to generate candidate invariants. Since the focus here is on location `L`, we refer to these as *L-states* and we distinguish those that are *observed* by instrumentation on a program run. It is these observed concrete L-states that form the basis for all dynamic invariant inference techniques.

On eight hand-selected set of inputs that seek to expose diverse concrete L-states, running Daikon [2] on this example results in very simple invariants, e.g., $\texttt{y1} \geq 0$, $\texttt{x2} \geq 2$. These are clearly much weaker than the desired invariant for this example. Moreover, the invariant on `x2` is actually spurious since clearly 1 can be passed as the second input

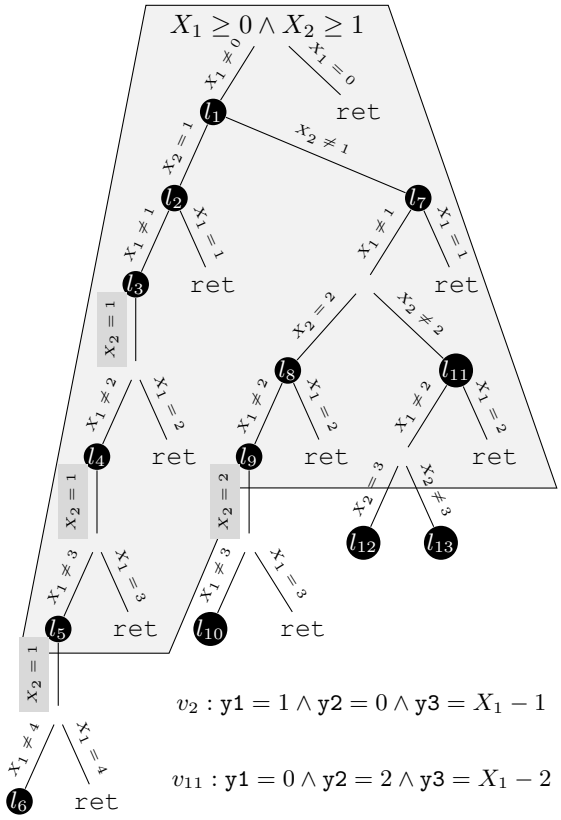

Fig. 2. Symbolic Execution Tree and Symbolic L-states

which will reach `L`. Applying the more powerful DIG [12] invariant generator, permits the identification of the desired invariant, but it too will yield the spurious $\texttt{x2} \geq 2$ invariant. Spurious invariants are a consequence of the diversity and representativeness of the inputs used, and the L-states that are observed. Leveraging symbolic states can help address this weakness.

A. Generating a symbolic state space

Figure 2 depicts a tree resulting from a depth-bounded symbolic execution of the example. The gray region includes paths limited to at most 5 branches; in this setting depth is a semantic property and syntactic branches with only a single infeasible outcome are not counted, e.g., the branches with labels enclosed in gray boxes. We denote the unknown values of program inputs using variables X_i and return points with `ret`.

The states at location `L` are denoted l_i in the figure. An observed symbolic L-state, l_i, is defined by the conjunction of the path-condition, i.e., the set of constraints on the tree-path to the state, and v_i, a constraint that encodes the values of local variables in scope at `L`. For example, the symbolic state l_2 is defined as $(X_2 = 1 \wedge X_1 \neq 0 \wedge X_1 \geq 0 \wedge X_2 \geq 1) \wedge (\texttt{y1} = 0 \wedge \texttt{y2} = 2 \wedge \texttt{y3} = X_1 - 2)$.

As is typical in symbolic execution, it is possible to increase the depth-bound and generate additional states, e.g., l_6, l_{10}, l_{12}, and l_{13} which all appear at a depth of 6 branches.

There are several properties of symbolic states that make them useful as a basis for efficient inference of invariants:

a) Symbolic states are expressive: Dynamic analysis has to observe many concrete L-states to obtain useful results. Many of those states may be equivalent from a symbolic perspective. A symbolic state, like l_2, encodes a potentially infinite set of concrete states, e.g., $X_1 > 0 \wedge X_2 = 1$. Invariant generation algorithms can exploit this expressive power to account for the generation and refutation of candidate invariants from a huge set of concrete states by processing a single symbolic state.

b) Symbolic states are relational: Symbolic states encode the values of program variables as expressions over free-variables capturing program inputs, i.e., X_i. This permits relationships between variables to be gleaned from the state. For example, state l_2 represents the fact that `y3` $<$ `x1` for a large set of inputs.

c) Symbolic states can be reused: Invariant generation has to infer or refute candidate invariants relative to the set of observed concrete L-states. This can grow in cost as the product of the number of candidates and the size of number of observed states. A disjunctive encoding of observed symbolic L-states, $\bigvee_{i\in[1-13]} l_i$, can be constructed a single time and reused for each of the candidate invariants, which can lead to performance improvement.

d) Symbolic states form a sufficiency test: The diversity of symbolic L-states found during depth-bounded symbolic execution combined with the expressive power of each of those states provides a rich basis for inferring strong invariants. We conjecture that for many programs a sufficiently rich set of observed L-states for invariant inference will be found at relatively shallow depth. For example, the invariants generated and not refuted by the disjunction of L-states at depth 5, $L_{k=5} = \{l_1, l_2, l_3, l_4, l_5, l_7, l_8, l_9, l_{11}\}$, is the same for those at depth 6, $\bigvee_{i\in[1-13]} l_i$. Consequently, we explore an adaptive and incremental approach that increases depth only when new L-states lead to changes in candidate invariants.

B. SymInfer in action

SymInfer will invoke a symbolic executor to generate a set of symbolic L-states at depth k, e.g., $k = 5$ in our example for the gray region. SymInfer then forms a small population of concrete L-states, using symbolic L-states, to generate a set of candidate invariants using DIG. DIG produces three invariants at L for this example: $\texttt{y1}\cdot\texttt{y2}\cdot\texttt{y3} = 0$, $\texttt{x2}\cdot\texttt{y1}-\texttt{x1}+\texttt{y2}+\texttt{y3} = 0$, and $\texttt{x1}\cdot\texttt{y3}-12\cdot\texttt{y1}\cdot\texttt{y3}-\texttt{y2}\cdot\texttt{y3}-\texttt{y3}^2 = 0$. SymInfer attempts to refute these invariants by using the full expressive power of the observed L-states to determine if all of the represented concrete states are consistent with the invariant. It does this by calling a SAT solver to check implications such as $\bigvee_{l\in L_{k=5}} l \Rightarrow (\texttt{y1}\cdot\texttt{y2}\cdot\texttt{y3} = 0)$. This refutes the first and third candidate invariant.

SymInfer then seeks additional L-states by running symbolic execution with a deeper bound, $k = 6$. While this process produces an additional 4 states to consider, none of those can refute the remaining invariant candidate. Thus, SymInfer terminates and produces the desired invariant.

```
void pldi_fig2(int M, int N, int P){
  assert (0 ≤ M && 0 ≤ N && 0 ≤ P);
  int i = 0, j = 0, k = 0;
  int t = 0; //counter variable
  while(i < N){//loop 1
    j = 0; t++;
    while(j < M){//loop 2
      j++; k = i; t++;
      while (k < P){// loop 3
        k++; t++;
      }
      i = k;
    }
    i++;
  }
  // [L]
}
```

Fig. 3. A program that has several nonlinear complexity bounds.

III. Dynamically Infer Numerical Invariants

A. Numerical Invariants

We consider invariants describing relationships over *numerical* program variables such as $x \leq y, 0 \leq idx \leq |arr| - 1, x + 2y = 100$. These numerical invariants have been used to verify program correctness, detect defects, establish security properties, synthesize programs, recover formal specifications, and more [2], [4], [13], [17]–[22]. A particularly useful class of numerical invariants involves *nonlinear* relations, e.g., $x \leq y^2, x2 \cdot y1 + y2 + y3 = x1$. While more complex these arise naturally in many safety-critical applications [13], [23].

In addition to capturing program semantics (e.g., as shown in Section II), nonlinear invariants can characterize the computational complexity of a program. Figure 3 shows a program, adapted from Figure 2 of [24], with nontrivial runtime complexity. At first, this program appears to take $O(NMP)$ due to the three nested loops. But closer analysis shows a more precise bound $O(N + NM + P)$ because the innermost loop 3, which is updated each time loop 2 executes, changes the behavior of the outer loop 1.

When analyzing this program, SymInfer discovers a complex nonlinear invariant over the variables P, M, N and t (a temporary variable used to count the number of loop iterations) at location L (program exit):

$$P^2Mt + PM^2t - PMNt - M^2Nt - PMt^2 + MNt^2 + PMt - PNt - 2MNt + Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0.$$

This nonlinear (degree 4) equality looks very different than the expected bound $N + NM + P$ or even NMP. However, when solving this equation (finding the roots of t), we obtain three solutions that describe the exact bounds of this program:

$$\begin{aligned} t &= 0 && \text{when} \quad N = 0, \\ t &= P + M + 1 && \text{when} \quad N \leq P, \\ t &= N - M(P - N) && \text{when} \quad N > P. \end{aligned}$$

These results give more precise bounds than the given bound $N + MN + P$ in [24].

input : terms, states
output: equalities among terms

1 eqInvs $\leftarrow \emptyset$
2 template $\leftarrow$ `createTemplate`(terms)
3 eqts $\leftarrow$ eqts $\cup$ `instantiate`(template, states)
4 sols $\leftarrow$ `solve`(eqts)
5 eqInvs = `extractEqts`(sols, terms)
6 **return** eqInvs

Fig. 4. `inferEqts`: DIG's algorithm for finding candidate equalities

B. Inferring Invariants using Concrete States

To infer numerical invariants, SymInfer uses the algorithms in DIG [12]. For numerical invariants, DIG finds (potentially nonlinear) equalities and inequalities. Like other dynamic analysis tools, DIG generates *candidate* invariants that only hold over observed concrete L-states.

1) Nonlinear Equalities: To generate nonlinear equality invariants, DIG uses *terms* to represent nonlinear information from the given variables up to a certain degree. For example, the set of 10 terms $\{1, x, y, z, xy, xz, yz, x^2, y^2, z^2\}$ consist of all monomials up to degree 2 over the variables $\{x, y, z\}$.

DIG then applies the steps shown in Figure 4 to generate equality invariants over these terms using concrete states observed at location L, and returns a set of possible equality relations among those terms. First, we use the input terms to form an equation *template* $c_1t_1 + c_2t_2 \cdots + c_nt_n = 0$, where and t_i are terms and c_i are real-valued unknowns to be solved for (line 2). Next, we instantiate the template with concrete states to obtain concrete equations (line 3). Then we use a standard equation solver to solve these equations for the unknowns (line 4). Finally we combine solutions for the unknowns (if found) with the template to obtain equality relations (line 5).

2) Octagonal Inequalities: DIG uses various algorithms to infer different forms of inequality relations. We consider the *octagonal* relations of the form $c_1v_1 + c_2v_2 \leq k$ where v_1, v_2 are variables and $c_i \in \{-1, 0, 1\}$ and k is real-valued. These relations represent linear inequalities among program variables, e.g., $x \leq y, -10 \leq x - y \leq 20$.

To infer octagonal invariants from concrete states $\{(x_1, y_1), \dots\}$, we compute the upper and lowerbounds:

$$\begin{aligned} u_1 = \max(x_i), l_1 = \min(x_i), \\ u_2 = \max(y_i), l_2 = \min(y_i), \\ u_3 = \max(x_i - y_i), l_3 = \min(x_i - y_i), \\ u_4 = \max(x_i + y_i), l_4 = \min(x_i + y_i) \end{aligned}$$

and form a set of 8 (octagonal) relations $\{u_1 \geq x \geq l_1, u_2 \geq y \geq l_2, u_3 \geq x - y \geq l_3, u_4 \geq x + y \geq l_4\}$.

Although computing octagonal inequalities is very efficient (linear in the number of concrete states), the candidate results are likely spurious because the upper and lower bound values might not be in the observed concrete states. SymInfer deals with such spurious invariants using a CEGIR approach described in Section IV.

IV. CEGIR Algorithms using Symbolic States

The behavior of a program at a location can be precisely represented by the set of all possible values of the variables in scope of that location. We refer to these values as the *concrete states* of the program. Figure 1 shows several concrete states observed at location L when running the program on inputs $(x1 = 15, x2 = 2)$ and $(x1 = 4, x2 = 1)$.

The set of all concrete states is the most precise representation of the relationship between variables at a program location, but it is potentially infinite and thus is difficult to use or analyze. In contrast, invariants capture program behaviors in a much more compact way. For the program in Figure 1 invariants at location L include: $0 \leq x1, 1 \leq x2, 0 \leq y2 + y3, x2 \cdot y1 + y2 + y3 = x1, \dots$ The most useful at L is $x2 \cdot y1 + y2 + y3 = x1$, which describes the semantics of integer division. The inequality $0 \leq y2 + y3$ is also useful because it asserts that the remainder is non-negative.

Dynamic invariant generation techniques, like Daikon and DIG, use concrete program states as inputs to compute useful invariants. We propose to compute invariants from the *symbolic states* of a program. Conceptually, symbolic states serve as an intermediary representation between a set of concrete program states and an invariant that might be inferred from those concrete states.

We assume a fixed and known set of variables in scope at a given location in a program. Moreover, we assume variables are indexed and that for an index i, $var(i)$ is a canonical name for that variable. Invariants will be inferred over these named variables. This is straightforward for locals and parameters, but permits richer naming schemes for other memory locations.

We write a set of appropriately typed values for those variables as $\vec{v} \equiv \langle v_1, v_2, \dots, v_n \rangle$, where the indexing corresponds to that of variables. Undefined variables have a $\perp$ value and the ith value is written $\vec{v}[i]$. A *concrete state* is $(l, \vec{v})$ where control is at location l and program variables have the values given by $\vec{v}$.

Let I be a set free-variables that denote the undefined input values of a program. A *symbolic value* is an expression written using constants, elements of I, and the operators available for the value's type. We write a sequence of symbolic values as $\vec{e} \equiv \langle e_1, e_2, \dots, e_n \rangle$.

Definition 1. *A symbolic state is $(l, \vec{e}, c)$ where control is at location l, c is a logical formula written over I, and a program variable takes on the corresponding concrete values that are consistent with c and symbolic value. The semantics of a symbolic state is:*

$$[\![(l, \vec{e}, c)]\!] = \{(l, \vec{v}) \mid SAT((\bigwedge_i \vec{v}[i] = \vec{e}[i]) \wedge c)\}$$

The role of c in a symbolic state is to define the constraints between variables, for example, that may be established on execution paths reaching l—a path condition.

A. Using Symbolic States

Symbolic states can help invariant generation in many ways. We describe two concrete techniques using symbolic states

input : prog P, L, number of states n, depth d
output: cstates

1 block $\leftarrow false$
2 cstates $\leftarrow \emptyset$
3 sstates $\leftarrow$ symex.`getStatesAt`(P, L, d)
4 **foreach** $s \in$ sstates **do**
5 **if** $SAT(s.c)$ **then**
6 $\vec{i} \leftarrow$ `getModel()`
7 cstates $\leftarrow$ cstates $\cup$ $(L, \texttt{eval}(s.\vec{e}, \vec{i}))$
8 block $\leftarrow$ block $\vee$ $(\bigwedge_{i \in I} \vec{i}[i] = i)$
9 **while** |cstates| $< n$ **do**
10 $s \leftarrow$ `choose`(sstates)
11 **if** $SAT(s.c \wedge \neg$block$)$ **then**
12 $\vec{i} \leftarrow$ `getModel()`
13 cstates $\leftarrow$ cstates $\cup$ $(L, \texttt{eval}(s.\vec{e}, \vec{i}))$
14 block $\leftarrow$ block $\vee$ $(\bigwedge_{i \in I} \vec{i}[i] = i)$
15 **return** cstates

Fig. 5. `genStates`: generate concrete states from symbolic states

to generate diverse concrete states and to verify candidate invariants.

1) Bootstrapping DIG with Concrete States: Our method generates candidate invariants using existing state of the art concrete state-based invariant inference techniques like DIG. In this application we need only use a small number of concrete states to bootstrap the algorithms to generate a diverse set of candidate invariants since symbolic states will be used to refute spurious invariants. In prior work [10], [12], fuzzing was used to generate inputs and that could be used here as well, but we can also exploit symbolic states.

Figure 5 shows how we use symbolic states to generate a diverse set of concrete states—at least one for each symbolic state. It first generates the set of symbolic L-states reachable depth less than or equal to d (line 3); note that these states can be cached and reused for a given P and L.

The loop on line 4 considers each such state, checks the satisfiability of the states path condition, c, and then extracts the model from the solver. We encode the model as a sequence, $\vec{i}$, indexed by the name of a free input variables. The symbolic state is then evaluated by the binding of concrete values to input variables in the model. This produces a concrete state which is accumulated. A conjunction of constraints equating the values of the model, $\vec{i}$, and the names of inputs, I, is added to the blocking clause for future state generation.

The loop on line 9 generates additional concrete states up to the requested number, n. This process will randomly `choose` a symbolic state and then call the SAT solver to generate a solution that has not already been computed; here $\vec{i}$ is converted to a conjunction of equality constraints between input variables and values from a model. When a solution is found, we use the same processing as in lines 6-7 to create a new concrete state.

2) Symbolic States as a "Verifier": Figure 6 shows how symbolic states are used to verify, or refute, a property. The

input : prog P, loc L, prop p, clauses to block
output: counterexample cex

1 p.isInv $\leftarrow$ unknown
2 result $\leftarrow$ unknown
3 result$'$ $\leftarrow$ unknown
4 cex $\leftarrow \emptyset$
5 $k \leftarrow 10$ //default depth
6 **while** true **do**
7 sstates $\leftarrow$ symex.`getStatesAt`(P, L, k)
8 vc $\leftarrow$ $(\bigvee_{s \in \text{sstates}} (s.c \wedge \bigwedge_i var(i) = s.\vec{e}[i])$
9 vc $\leftarrow$ vc $\wedge \neg(\bigvee$ block$)$
10 result$'$ $\leftarrow$ `SAT`$(\neg(\text{vc} \Rightarrow p)$
11 **if** result$'$ $\equiv$ result **then**
12 $k \leftarrow k - 1$
13 `break`
14 result $\leftarrow$ result$'$
15 **if** result$'$ $\equiv$ sat **then**
16 p.isInv $\leftarrow$ false
17 cex $\leftarrow$ `getModel()`
18 `break`
19 **else if** result$'$ $\equiv$ unsat **then**
20 p.isInv $\leftarrow$ true
21 **else if** result$'$ $\equiv$ unknown **then**
22 p.isInv $\leftarrow$ unknown
23 $k \leftarrow k + 1$
24 **return** cex

Fig. 6. `verify`: check a candidate property using symbolic states

algorithm obtains new symbolic states when it is determined that they increase the accuracy of the verification.

Symbolic states are obtained from a symbolic execution engine. There are potentially an infinite number of symbolic states at a location, but most existing symbolic execution tools have the ability to perform a depth-limited search. We wrap the symbolic execution engine to just return the symbolic L-states encountered during search of a given depth (`getStatesAt`).

The number of symbolic states varies with depth. A low depth means few states. Few states will tend to encode a small set of concrete L-states, which limits verification and refutation power. Few states will also tend to produce a smaller and faster to solve verification condition. To address this cost-effectiveness tradeoff, rather than try to choose an optimal depth, our algorithm computes the lowest depth that yields symbolic states that change verification outcomes. In essence, the algorithm adaptively computes a good cost-effectiveness tradeoff for a given program, location of interest, and invariant.

The algorithm iterates with each iteration considering a different depth, k. The body of the each iteration (lines 7 – 23) works as follows. It extract a set of symbolic states for the current depth using symbolic execution (line 7); note this can be done incrementally to avoid re-exploring the program's state space using techniques like [25]. It then formulates a verification condition out of three components. (1) For each symbolic state, it constructs the conjunction of its path condition, c, with constraints encoding equality constraints between variables and their symbolic values, $\vec{e}$; these per-state conjunctions are then disjoined. This expresses the set of

concrete L-states corresponding to all of the symbolic states. (2) The negation of the disjunction of the set of states that are to be blocked is formed. These components are conjoined, which serves to eliminate the concrete L-states that are to be blocked. (3) If the resulting formula implies a candidate p then that candidate is consistent with the set of symbolic states. We use a SAT solver to check the negation of this implication.

The solver can return `sat` which indicates that the property is not an invariant (lines 15 – 18). The solver is also queried for a model which is a sample state that is inconsistent with the proposed invariant. This counterexample state is saved so that the inference algorithm can search for invariants that are consistent with it. The solver can also return `unsat` indicating the property is a true invariant; at least as far as the algorithm can determine given the symbolic states at the current depth. Finally, the solver can also return unknown, indicating it cannot determine whether the given property is true or false.

For the latter two cases, we increment the depth and explore a larger set of symbolic states generated from a deeper symbolic execution. Lines 10 – 14 work together to determine when increasing the depth does not influence the verification. In essence, they check to see whether the same result is computed at adjacent depths and if so, they revert to the shallower depth and return.

B. A CEGIR approach using symbolic states

CounterExample Guided Invariant Generation (CEGIR) techniques consist of a guessing component that infers candidate invariants and a checking component that verifies the candidate solutions. If the candidate is invalid, the checker produces *counterexamples*, i.e., concrete states that are not consistent with the candidate invariant. The guessing process incorporates the generated counterexamples so that any new invariants account for them. Alternation of guessing and checking repeats until no candidates can be disproved.

SymInfer integrates symbolic traces into two CEGIR algorithms to compute candidate invariants. These algorithms use the inference techniques described in Section III for equality and inequality invariants.

1) Nonlinear Equalities: Figure 7 defines our CEGIR algorithm for computing non-linear equality invariants. It consists of two phases: an initial invariant candidate generation phase and then an iterative invariant refutation and refinment phase.

Lines 5 – 7 define the initial generation phase. As as described in Section III-B1, we first create terms to represent nonlinear polynomials (line 5). Because solving for n unknowns requires at least n unique equations, we need to generate a sufficient set of concrete L-states (line 6). This can either be realized through fuzzing an instrumented version of the program that records concrete L-states or, as described in Figure 5, one can use symbolic L-states to generate them.

The initial candidate set of invariants is iteratively refined on lines 8 – 18. The algorithm then refutes or confirms them using symbolic states as described in Figure 6. Any property that is proven to hold is recorded in $invs$ and counterexample states,

input : program P, location L, degree d
output: nonlinear equalities up to deg d at L

```
1  states ← ∅
2  invs ← ∅
3  block ← ∅
4  vars ← extractVars(P, L)
5  terms ← createTerms(vars, d)
6  states ← genStates(P, L, |terms|)
7  candidates ← inferEqts(terms, states)
8  while candidates ≠ ∅ do
9      cexs ← ∅
10     foreach p ∈ candidates do
11         newcexs ← verify(P, L, p, block)
12         cexs ← cexs ∪ newcexs
13         if p.isInv then invs ← invs ∪ {p}
14     if cexs ≡ ∅ then break
15     block ← block ∪ cexs
16     states ← states ∪ cexs
17     newcandidates ← inferEqts(termsstates)
18     candidates ← newcandidates − invs
19 return invs
```

Fig. 7. CEGIR algorithm for finding equalities.

$cexs$, are accumulated across the set of properties. Generated counterexample states are also blocked from contributing to the verification process.

If no property generated counterexample states, then the algorithm terminates returning the verified invariants. The counterexamples are added to the set of states that are used to infer new candidate invariants; this ensures that new invariants will be consistent with the counterexample states (line 16). These new results may include some already proven invariants, so we remove those from the set of candidates considered in the next round of refinement.

2) Octagonal Inequalities: Our next CEGIR algorithm uses a divide and conquer approach to compute octagonal inequalities. Given a term t, and an interval range $[minV, maxV]$, we compute the smallest integral upperbound k of t by repeatedly dividing the interval into halves that could contain k. The use of an interval range $[minV, maxV]$ allows us to exclude terms ranges are too large (or that do not exist). For example, if we check $t > maxV$ and it holds then we will not compute the bound of t (which is strictly larger than $maxV$).

We start by checking a guess that $t \leq midV$, where $midV = \lceil \frac{maxV+minV}{2} \rceil$. These checks are performed by formulating a verification condition from symbolic states in a manner that is analogous to Figure 7. If this holds, then k is at most $midV$ and we tighten the search to a new interval $[minV, midV]$. Otherwise, we obtain counterexample with t having some value c, where $c > midV$. We then tighten the search to a new interval $[c, maxV]$. In either case, we repeat the guess for k using an interval that is half the size of the previous one. The search stops when $minV$ and $maxV$ are the same or their difference is one (in which case we return the smaller value if t is less than or equal both).

To find octagonal invariants over a set of variables, e.g., $\{x, y, z\}$, we apply this method to find upperbounds of the terms $\{x, -x, y, -y, \ldots, y + z, -y - z\}$. Note that we obtain

both lower and upperbound using the same algorithm because the upperbound for t essentially lowerbound of $-t$ since all computations are reversed for $-t$.

SymInfer reuses the symbolic states from the inference of equalities to formulate verification conditions for inequalities. This is another example of how reuse speeds up inference.

V. Implementation and Evaluation

We implemented SymInfer in Python/SAGE [26]. The tool takes as input a Java program with marked target locations and generates invariants at those locations. We use Symbolic PathFinder (SPF) [9] to extract symbolic states for Java programs and the Z3 SMT Solver [20] to check and produce models representing counterexamples. We also use Z3 to check and remove redundant invariants.

SymInfer currently supports equality and inequality relations over numerical variables. For (nonlinear) equalities, SymInfer uses techniques from DIG to limit the number of generated terms. This allows us, for example, to infer equalities up to degree 5 for a program with 4 variables and up to degree 2 for program with 12 variables. For octagonal invariants, we consider upper and lower bounds within the range $[-10, 10]$; we rarely observe inequalities with large bounds. SymInfer can either choose random values in a range, $[-300, 300]$ by default, for bootstrapping, or use the algorithm in Figure 5. All these parameters can be changed by SymInfer's user; we chose these values based on our experience.

A. Research Questions

To evaluate SymInfer, we consider three research questions:

1) Is SymInfer effective in generating nonlinear invariants describing complex program semantics and correctness?
2) Can SymInfer generate expressive invariants that capture program runtime complexity?
3) How does SymInfer perform relative to PIE, a state-of-the-art invariant generation technique?

To investigate these questions, we used 3 benchmark suites consist of 92 Java programs (described in details in each section). These programs come with known or documented invariants. Our objective is to compare SymInfer's inferred invariants against these documented results. To compare invariants, we used Z3 to check if the inferred results imply the documented ones. We use a script to run SymInfer 11 times on each program and report the median results. The scripts automatically terminates a run exceeding 5 minutes. The experiments reported here were performed on a 10-core Intel i7 CPU 3.0GHZ Linux system with 32 GB of RAM.

B. Analyzing Program Correctness

In this experiment, we use the NLA testsuite [12] which consists of 27 programs implementing mathematical functions such as `intdiv`, `gcd`, `lcm`, `power`. Although these programs are relatively small (under 50 LoCs) they contain nontrivial structures such as nested loops and nonlinear invariant properties. To the best of our knowledge, NLA contains the largest number of programs containing nonlinear arithmetic. These programs have also been used to evaluate other numerical invariant systems [12], [27], [28].

TABLE I
Experimental results for 27 programs in the NLA testsuite. ✓ indicates when SymInfer generates results sufficiently strong enough to prove known invariants.

Prog	Desc	Locs	V, T, D	Invs	Time (s)	Correct
cohendiv	int div	2	6,3,2	10	21.05	✓
divbin	int div	2	5,3,2	11	58.97	✓
manna	int div	1	5,4,2	6	35.33	✓
hard	int div	2	6,3,2	6	29.40	✓
sqrt	square root	1	4,4,2	5	20.03	✓
dijkstra	square root	2	5,7,3	16	93.01	✓
freire1	square root	1	-	-	-	-
freire2	cubic root	1	-	-	-	-
cohencu	cubic sum	1	5,5,3	4	21.90	✓
egcd1	gcd	1	8,3,2	14	122.22	✓
egcd2	gcd	2	-	-	-	-
egcd3	gcd	3	-	-	-	-
prodbin	gcd, lcm	1	5,3,2	7	56.17	✓
prod4br	gcd, lcm	1	6,3,3	9	84.37	✓
knuth	product	1	-	-	-	-
fermat1	product	3	5,6,2	17	60.26	✓
fermat2	divisor	1	5,6,2	8	36.83	✓
lcm1	divisor	3	6,3,2	24	248.17	✓
lcm2	divisor	1	6,3,2	7	34.17	✓
geo1	geo series	1	4,4,2	8	158.27	✓
geo2	geo series	1	4,4,2	9	147.75	✓
geo3	geo series	1	-	-	-	-
ps2	pow sum	1	3,3,2	3	18.39	✓
ps3	pow sum	1	3,4,3	3	19.69	✓
ps4	pow sum	1	3,4,4	3	19.92	✓
ps5	pow sum	1	3,5,5	3	46.19	✓
ps6	pow sum	1	3,5,6	3	41.19	✓

These NLA programs come with known program invariants at various program locations (e.g., mostly nonlinear equalities for loop invariants and postconditions). For this experiment, we evaluate SymInfer by finding invariants at these locations and comparing them with known invariants.

Results: Table I shows the results of SymInfer for the 27 NLA programs. Column **Locs** show the number of locations where we obtain invariants. Column **V,T,D** shows the number of variables, terms, and highest degree from these invariants. Column **Invs** shows the number of discovered equality and inequality invariants. Column **Time** shows the total time in seconds. Column **Correct** shows if the obtained results match or imply the known invariants.

For 21/27 programs, SymInfer generates correct invariants that match or imply the known results. In most cases, the discovered invariants match the known ones exactly. Occasionally, we obtain results that are equivalent or imply the known results. For example, for `sqrt`, for some runs we obtained the documented equalities $t = 2a + 1, s = (a + 1)^2$, and for other runs we obtain $t = 2a + 1, t^2 - 4s + 2t = -1$, which are equivalent to $s = (a+1)^2$ by replacing t with $2a+1$. We also obtain undocumented invariants, e.g., SymInfer generates the postconditions $x = qy + r, 0 \leq r, r \leq x, r \leq y - 1$ for `cohendiv`, which computes the integer division result of two integers $q = x \div y$, The first invariant is known and describes the precise semantics of integer division: the dividend x is the divisor y times the quotion q plus the remainder r. The

TABLE II
EXPERIMENTAL RESULTS FOR COMPUTING PROGRAMS' COMPLEXITIES. ✓: SYMINFER GENERATES THE EXPECTED BOUNDS. ✓*: PROGRAM WAS SLIGHTLY MODIFIED TO ASSIST THE ANALYSIS. ✓✓: SYMINFER OBTAINS MORE PRECISE BOUNDS THAN REPORTED RESULTS.

Prog	V, T, D	Invs	Time (s)	Bound
cav09_fig1a	2,5,2	1	12.41	✓
cav09_fig1d	2,5,2	1	12.44	✓
cav09_fig2d	3,2,2	3	58.40	✓
cav09_fig3a	2,2,2	3	8.75	✓
cav09_fig5b	3,5,2	6	49.44	✓*
pldi09_ex6	3,8,3	6	57.00	✓
pldi09_fig2	3,15,4	6	60.60	✓✓
pldi09_fig4_1	2,3,1	3	56.24	✓
pldi09_fig4_2	4,4,2	5	28.32	✓
pldi09_fig4_3	3,3,2	3	59.19	✓
pldi09_fig4_4	5,4,2	-	-	-
pldi09_fig4_5	3,4,2	3	103.70	✓*
popl09_fig2_1	5,12,3	2	50.86	✓✓
popl09_fig2_2	4,9,3	2	53.48	✓✓
popl09_fig3_4	3,4,3	4	58.62	✓
popl09_fig4_1	3,3,2	4	65.19	✓*
popl09_fig4_2	5,12,3	2	51.24	✓✓
popl09_fig4_3	3,3,2	5	31.57	✓
popl09_fig4_4	3,3,2	3	36.89	✓

other obtained inequalities were undocumented. For example, $r \geq 0$ asserts that the remainder r is non-negative and $r \leq x, r \leq y - 1$ state that r is at most the dividend x, but is strictly less than the divisor y. Our experience shows that SymInfer is capable of generating many invariants that are unexpected yet correct and useful.

SymInfer did not find invariants for 6/27 programs (marked with "-" in Table I). For `egcd2`, `egcd3`, the equation solver used in SAGE takes exceeding long time for more than half of the runs. For `geo3`, we obtained the documented invariants and others, but Z3 stops responding when checking these results. `freire1` and `freire2` contain floating point arithmetic, which are currently not supported by SymInfer. SPF failed to produce symbolic states for `knuth` for any depth we tried. This program invokes a library function `Math.sqrt` and SPF does not know the semantics of this function and thus fails to provide useful symbolic information. For `egcd2`, `egcd3`, and `geo3`, SymInfer times out after 5 minutes, and for `freire1`, `freire2`, and `knuth`, it exits upon encountering the unsupported feature.

C. Analyzing Computational Complexity

As shown in Section III, nonlinear invariants can represent precise program runtime complexity. More specifically, we compute the roots of nonlinear relationships to obtain disjunctive information (e.g., $x^2 = 4 \Rightarrow (x = 2 \vee x = -2)$, which apture different and precise complexity bounds of programs.

To further evaluate SymInfer on discovering program complexity, we collect 19 programs, adapted from existing static analysis techniques specifically designed to find runtime complexity [24], [29], [30][1]. These programs, shown in Table II, are small, but contain nontrivial structures and represent examples from Microsoft's production code [24]. For this experiment, we instrument each program with a fresh variable t representing the number of loop iterations and generate postconditions over t and input variables (e.g., see Figure 3).

Results: Table II shows the median results of SymInfer from 11 runs. Column **Bound** contains a ✓if we can generate invariants matching the bounds reported in the respective work, and ✓✓if the discovered invariants represent more precise bounds than the reported ones.A ✓* indicates when the program was modified slightly to help our analysis—described below.

For 18/19 programs, SymInfer discovered runtime complexity characterizations that match or improve on reported results. For `cav09_fig1a`, we found the invariant $mt - t^2 - 100m + 200t = 10000$, which indicates the correct bound $t = m + 100 \vee t = 100$. For these complexity analyses, we also see the important role of combining both inequality and equality relations to produce informative bounds. For `popl09_fig3_4`, SymInfer inferred nonlinear equality showing that $t = n \vee t = m$ and inequalities asserting that $t \geq n \wedge t \geq m$, together indicating that $t = \mathsf{max}(n, m)$, which is the correct bound for this program. In four programs, SymInfer obtains better bounds than reported results. The `pldi_fig2` programs showing in Figure 3 is a concrete example where the obtained three bounds are strictly less than the given bound.

For several programs we needed some manual instrumentation or inspections to help the analysis. For `popl09_fig4_1` we added the precondition asserting the input m is nonnegative. For `pldi09_fig4_5`, we obtained nonlinear results giving three bounds $t = n - m$, $t = m$, and $t = 0$, which establish the reported upperbound $t = max(0, n - m, m)$. For `pldi09_fig4_4`, we obtained invariants that are insufficient to show the reported bound. However, if we create a new term representing the quotient of an integer division of two other variables in the program, and obtain invariants over that term, we obtain more precise bounds than those reported.

D. Comparing to PIE

We compare SymInfer to the recent CEGIR-based invariant tool PIE [6]. PIE aims to verify annotated relations by generating invariants based on the given assertions. In contrast, SymInfer generates invariants at given locations without given assertions or postconditions. We use the HOLA benchmarks [31], adapted by the PIE developers. These programs are annotated with various assertions representing loop invariants and postconditions. This benchmark consists of 49 small programs, but contain nontrivial structures including nested loops or multiple sequential loops. These programs, shown in Table III, have been used as benchmarks for other static analysis techniques [32]–[34].

For this experiment, we first run PIE and record its run time on proving the annotated assertions. Next, we removed the assertions in the programs and asked SymInfer to generate invariants at those locations. Our objective is to compare SymInfer's discovered invariants with the annotated assertions. Because these HOLA programs only consist of asser-

[1]We remove nondeterministic features in these programs because SymInfer assumes determinstic behaviors.

TABLE III
SYMINFER RUN ON HOLA BENCHMARKS. ✓: PRODUCE SUFFICIENTLY STRONG RESULTS TO PROVE ASSERTIONS. ○: FAIL TO MAKE SUFFICIENTLY STRONG INVARIANTS.

Benchmark	PIE time (s)	SymInfer time (s)	Correct
H01	21.88	3.69	○
H02	36.12	3.36	✓
H03	56.28	23.96	✓
H04	19.11	3.12	✓
H05	25.19	3.76	✓
H06	61.98	4.56	✓
H07	-	4.58	✓
H08	19.02	4.33	✓
H09	-	19.66	✓
H10	24.6	4.25	✓
H11	27.95	5.13	✓
H12	44.52	14.60	✓
H13	-	3.99	✓
H14	25.98	4.07	✓
H15	48.30	4.20	✓
H16	33.19	4.99	✓
H17	53.36	3.03	✓
H18	21.70	5.69	✓
H19	-	5.05	✓
H20	331.93	29.45	✓
H21	25.65	18.99	○
H22	25.40	4.50	✓
H23	23.40	4.90	✓
H24	51.22	-	-
H25	-	4.31	✓
H26	87.64	5.55	✓
H27	55.41	-	-
H28	22.16	6.37	✓
H29	58.82	6.80	✓
H30	33.92	4.42	○
H31	88.10	38.94	✓
H32	226.73	6.75	✓
H33	-	6.95	✓
H34	121.87	11.34	✓
H35	20.07	3.47	✓
H36	-	7.61	✓
H37	-	9.87	✓
H38	37.37	6.47	✓
H39	24.68	3.99	✓
H40	60.71	60.20	✓
H41	34.10	6.89	✓
H42	54.93	5.55	○
H43	21.16	5.34	✓
H44	31.92	13.67	✓
H45	84.00	5.39	✓
H46	27.56	6.21	✓

tions having linear relations, we ask SymInfer to only generate invariants up to degree 2 (quadratic relations can represent linear relations, e.g., $x^2 = 4 \Rightarrow x = 2 \vee x = -2$).

Results: Table III shows these obtained results from PIE and SymInfer. Column **PIE time** shows the time, in seconds, for PIE to run each program. Column **SymInfer time** shows the time, in seconds, for SymInfer to generate invariants for each program (the median of 11 runs). The "-" symbol indicates when PIE fails to prove the given assertions, e.g., because it generates invariants that are too weak. Column **Correct** shows whether SymInfer's generated invariants match or imply the annotated assertions and therefore prove these assertions. For this experiment we manually check the result invariants and use Z3 to compare them to the given assertions. A ✓indicates that the generated invariants match or imply the assertions. A ○ indicates that the generated invariants are not sufficiently strong to prove the assertions.

For 40/46 programs, SymInfer discovered invariants are sufficiently strong to prove the assertions. In most of these cases we obtained correct and stronger invariants than the given assertions. For example, for H23, SymInfer inferred the invariants $i = n, n^2 - n - 2s = 0, -i \leq n$, which imply the postcondition $s \geq 0$. For H29, we obtained the invariants $b + 1 = c, a + 1 = d, a + b \leq 2, 2 \leq a$, which imply the given postcondition $a + c = b + d$.

Surprisingly, SymInfer also found invariants that are precise enough to establish conditions under forms that are *not* supported by SymInfer. For example, H8 contains a postcondition $x < 4 \ \vee \ y > 2$, which has a disjunctive form of strict inequalities. SymInfer did not produce this invariant, but instead produced a correct and stronger relation $x \leq y$, which implies this condition. Many HOLA programs contain disjunctive (or conditional) properties, e.g., `if(c) assert (p);` where the property p only holds when the condition c holds (written $c \Rightarrow p$). For example, for H18, we obtained $fj = 100f$, which implies the conditional assertion $f \neq 0 \Rightarrow j = 100$. For H37, PIE failed to prove the postcondition `if (n > 0) assert(0 <= m && m < n);` which involves both conditional assertions and strict inequalities. For this program, SymInfer inferred 2 equations and 3 inequalities[2], which together establish the postcondition.

For 6/46 programs, SymInfer either failed to produce invariants (2 programs marked with "-") or discovered invariant that are not strong enough to prove the given assertions (4 programs marked with ○). For both H24 and H27, Z3 stops responding when checking the inferred results and the run were terminated after 5 minutes. For H01, we found the invariant $x = y$, which is not sufficiently to establish the postcondition $y \leq 1$. For H27, SymInfer found no relation involving the variable c to prove the assertion $c \geq 0$.

Summary: These preliminary results show SymInfer generates expressive, useful, and interesting invariants describing the semantics and match documented invariants (21/27 NLA programs), discovers difficult invariants capturing precise and informative complexity bounds of programs (18/19 programs), and is competitive with PIE (40/46 HOLA programs). We also note that PIE, ICE, and iDiscovery (other CEGIR-based tools reviewed in Section VI), cannot find any of these high-degree nonlinear invariants found by SymInfer.

E. Threats to Validity

SymInfer's run time is dominated by computing invariants, more specifically solving hundred of equations for hundred of unknowns. The run time of DIG can be improved significantly by limiting the search to invariants of a given maximum degree rather than using the default setting. Verifying candidate invariants, i.e., checking implication using the Z3 solver, is much faster than DIG, even when multiple checks are performed at different depths. This shows an advantage of reusing symbolic states when checking new invariants.

[2] $m^2 = nx - m - x, mn = x^2 - x, -m \leq x, x \leq m + 1, n \leq x$

SymInfer encodes all symbolic states to into the Z3 verification condition. This results in complex formulas with large disjunctions that can make Z3 timeout. Moreover, depending on the program, SPF might not be able to generate all possible symbolic states. In such cases, SymInfer cannot refute candidate invariants and thus may produce *unsound* results. However, our experience shows that SPF, by its nature as a symbolic executor, turns out to be very effective in producing sufficient symbolic states, which effectively remove invalid candidates.

Finally, we reuse existing analysis tools, such as DIG and SPF, which provides a degree of assurance in the correctness of SymInfer, but our primary means of assuring internal validity was performing both manual and automated (SMT) checking of the invariants computed for all subject programs. While our evaluation uses a variety of programs from different benchmarks, these programs are small and thus do not represent large software projects. Their use does promote comparative evaluation and reproducibility of our results. We believe using symbolic states will allow for the generation of useful and complex invariants for larger software systems, in part because of the rapid advances in symbolic execution and SMT solving technologies and SymInfer leverages those advances.

VI. Related Work and Future Work

Daikon [2] is a well-known dynamic tool that infers candidate invariants under various templates over concrete program states. The tool comes with a large set of templates which it tests against observed concrete states, removing those that fail, and return the remaining ones as candidate invariants. DIG [12] is similar to Daikon, but focuses on numerical invariants and therefore can compute more expressive numerical relations than those supported by Daikon's templates.

PIE [6] and ICE [7] uses CEGIR to infer invariants to prove a given specification. To prove a property, PIE iteratively infers and refined invariants by constructing necessary predicates to separate (good) states satisfying the property and (bad) states violating that property. ICE uses a decision learning algorithm to guess inductive invariants over predicates separating good and bad states. The checker produces good, bad, and "implication" counterexamples to help learn more precise invariants. For efficiency, they focus on octagonal predicates and only search for invariants that are boolean combinations of octagonal relations. In general, these techniques focus on invariants that are necessary to prove a given specification and, thus, the quality of the invariants are dependent target specification.

NumInv [8] is a recent CEGIR tool that discovers invariants for C programs. The tool also uses DIG's algorithms to infer equality and inequality relations. For verification it instruments invariants into the program and runs the KLEE test-input generation tool [16]. KLEE does use a symbolic state representation internally, but this is inaccessible to NumInv. Moreover, KLEE is unaware of its use in this context and it recomputes the symbolic state space completely for each verification check, which is inefficient. For the experiments in Section V, SymInfer is comparable to NumInv in the quality of invariants produced, but SymInfer runs faster in spite of the fact that KLEE's symbolic execution of C programs is known to be faster than SPF's performance on Java programs. We credit this to the benefits of using symbolic states.

Similar to SymInfer, the CEGIR-based iDiscovery [5] tool uses SPF to check invariants. However, iDiscovery does not exploit the internal symbolic state representation of symbolic excution but instead runs SPF as a blackbox to check program assertions encoding candidate invariants. To speed up symbolic execution, iDiscovery applies several optimizations such as using the Green solver [35] to avoid recomputing the symbolic state space for each check. In contrast, SymInfer precomputes the full disjunctive SMT formula encoding the paths to the interested location once and reuses that formula to check candidate invariants. For dynamic inference, iDiscovery uses Daikon and thus has limited support for numerical invariants. For example, iDiscovery cannot produce the required nonlinear invariants or any relevant inequalities for the programs in Figures 1 and 3. Note that for programs involving non-numerical variables, Daikon/iDiscovery might be able to infer more invariants than SymInfer.

SymInfer is unlike any of the above in its reliance on symbolic states to bootstrap, verify and iteratively refine the invariant generation process. There are clear opportunities for significantly improving the performance of SymInfer and targeting different languages, such as C through the use of other symbolic executors. For example, generating symbolic states can be sped up for invariant inference by combining directed symbolic execution [36] to target locations of interest, memoized symbolic execution [25] to store symbolic execution trees for future extension, and parallel symbolic execution [37] to accelerate the incremental generation of the tree. Moreover, we can apply techniques for manipulating symbolic states in symbolic execution [16], [35] to significantly reduce the complexity of the verification conditions sent to the solver.

VII. Conclusion

We present SymInfer a method that uses symbolic encodings of program states to efficiently discover rich invariants over numerical variables at arbitrary program locations. SymInfer uses a CEGIR approach that uses symbolic states to generate candidate invariants and also to verify or refute, and iteratively refine, those candidates. Key to the success of SymInfer is its ability to directly manipulate and reuse rich encodings of large sets of concrete program states. Preliminary results on a set of 92 nontrivial programs show that SymInfer is effective in discovering useful invariants to describe precise program semantics, characterize the runtime complexity of programs, and verify nontrivial correctness properties.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Number 1617916.

REFERENCES

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, pp. 35–45, 2007.

[2] M. D. Ernst, "Dynamically detecting likely program invariants," Ph.D. dissertation, University of Washington, 2000.

[3] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Symposium on Operating Systems Principles*. ACM, 2009, pp. 87–102.

[4] W. Weimer, "Patches as better bug reports," in *Generative Programming and Component Engineering*. ACM, 2006, pp. 181–190.

[5] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *ISSTA*. ACM, 2014, pp. 362–372.

[6] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *PLDI*. ACM, 2016, pp. 42–56.

[7] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *POPL*. ACM, 2016, pp. 499–512.

[8] T. Nguyen, T. Antopoulos, A. Ruef, and M. Hicks, "A Counterexample-guided Approach to Finding Numerical Invariants," in *FSE*. ACM, 2017, pp. 605–615.

[9] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder," in *TACAS*. Springer-Verlag, 2007, pp. 134–138.

[10] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using Dynamic Analysis to Discover Polynomial and Array Invariants," in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 683–693.

[11] ——, "Using Dynamic Analysis to Generate Disjunctive Invariants," in *ICSE*. IEEE, 2014, pp. 608–619.

[12] ——, "DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants," *TOSEM*, vol. 23, no. 4, pp. 30:1–30:30, 2014.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The Astrée analyzer," in *ESOP*. Springer, 2005, pp. 21–30.

[14] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and synthesizing constant-resource implementations with types," in *Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 710–728.

[15] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *PLDI*. ACM, 2017, pp. 362–375.

[16] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*. USENIX Association, 2008, pp. 209–224.

[17] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN*. Springer, 2001, pp. 103–122.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[19] M. Das, S. Lerner, and M. Seigle, "ESP: path-sensitive program verification in polynomial time," in *PLDI*. ACM, 2002, pp. 57–68.

[20] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[21] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL*. ACM, 2006, pp. 42–54.

[22] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*. ACM, 2010, pp. 61–72.

[23] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *PLDI*. ACM, 2003, pp. 196–207.

[24] S. Gulwani, S. Jain, and E. Koskinen, "Control-flow refinement and progress invariants for bound analysis," in *PLDI*, 2009, pp. 375–385.

[25] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *ISSTA*. ACM, 2012, pp. 144–154.

[26] W. A. Stein *et al.*, "Sage Mathematics Software," 2017, http://www.sagemath.org.

[27] E. R. Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *Journal of Symbolic Computation*, vol. 42, no. 4, pp. 443–476, 2007.

[28] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data-driven approach for algebraic loop invariants," in *ESOP*. Springer, 2013, pp. 574–592.

[29] S. Gulwani, "SPEED: Symbolic complexity bound analysis," in *CAV*. Springer-Verlag, 2009, pp. 51–62.

[30] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "SPEED: precise and efficient static estimation of program computational complexity," in *POPL*. ACM, 2009, pp. 127–139.

[31] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, 2013, pp. 443–456.

[32] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[33] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *CAV*. Springer-Verlag, 2009, pp. 634–640.

[34] B. Jeannet, "Interproc analyzer for recursive programs with numerical variables," 2014, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.

[35] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, Reusing and Recycling Constraints in Program Analysis," in *FSE*. ACM, 2012, pp. 58:1–58:11.

[36] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS*. Springer-Verlag, 2011, pp. 95–111.

[37] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *ISSTA*. ACM, 2010, pp. 183–194.