Verification of Java Programs using Symbolic Execution and Invariant Generation

Corina S. Păsăreanu[1] and Willem Visser[2]

[1] Kestrel Technology, NASA Ames Research Center, Moffett Field, CA 94035, USA
pcorina@email.arc.nasa.gov
[2] RIACS/USRA, NASA Ames Research Center, Moffett Field, CA 94035, USA
wvisser@email.arc.nasa.gov

Abstract. Software verification is recognized as an important and difficult problem. We present a novel framework, based on symbolic execution, for the automated verification of software. The framework uses annotations in the form of method specifications and loop invariants. We present a novel iterative technique that uses invariant strengthening and approximation for discovering these loop invariants automatically. The technique handles different types of data (e.g. boolean and numeric constraints, dynamically allocated structures and arrays) and it allows for checking universally quantified formulas. Our framework is built on top of the Java PathFinder model checking toolset and it was used for the verification of several non-trivial Java programs.

1 Introduction

Model checking is becoming a popular technique for the verification of software [1,6,21,30], but it typically can only deal with closed systems and it suffers from the state-explosion problem. In previous work [23] we have developed a verification framework based on *symbolic execution* [24] and model checking that allows the analysis of complex software that take inputs from unbounded domains with complex structure, and helps combat state-space explosion. In that framework, a program is instrumented to add support for manipulating formulas and for systematic treatment of aliasing, so that to enable a standard model checker to perform symbolic execution of the program. The framework is built on top of the Java PathFinder model checker and it was used for test input generation and for error detection in complex Java programs, but it could not be used for proving properties of programs containing loops.

We present here a method that uses the symbolic execution framework presented in [23] for *proving* (light-weight) specifications of Java programs that contain loops. The method requires annotations in the form of method specifications and loop invariants. We present a novel iterative technique that uses invariant strengthening and approximation for discovering these loop invariants automatically. Our technique uniformly handles different types of constraints (e.g. boolean and numeric constraints, constraints on dynamically allocated structures and arrays) and it allows for checking universally quantified formulas. These formulas

are necessary for expressing properties of programs that manipulate unbounded data, such as arrays.

Our technique for loop invariant generation works backward from the property to be checked and has three basic ingredients: iterative invariant strengthening, iterative approximation and refinement. Symbolic execution is used to check that the current invariant is inductive: *the base case* checks that the current candidate invariant is true when entering the loop and the *induction step* checks that the current invariant is maintained by the execution of the loop body. Failed proofs of the induction step are used for iterative invariant strengthening, a process that may result in a (possibly infinite) sequence of candidate invariants. At each strengthening step, we further use a novel iterative approximation technique to achieve termination.

For strengthening step k, we use a (finite) set of relevant constraints called the *universe of constraints* U_k. The iterative approximation consists of a sequence of strengthening in which we drop all the constraints that are newly generated (and are not present in U_k). Since U_k is finite, this process is guaranteed to converge to an inductive approximate invariant that is a boolean combination of the constraints in U_k. The intuition here has similarities to *predicate abstraction* techniques [17], that perform iterative computations over a finite set of predicates (i.e. constraints). A failed *base case* proof can either indicate that there is an error in the program or that the approximation that we use at the current step is too strong, in which case we use *refinement*, that consists of enlarging the universe of constraints with new constraints that come from the next candidate invariant (computed at step $k+1$).

Loop invariant generation has received much attention in the literature, see e.g. [5, 8, 25, 29, 31]. Most of the methods presented in these papers were concerned with the generation of numerical invariants. A recent paper [13] describes a loop invariant generation method for Java programs that uses predicate abstraction. The method handles universally quantified specifications but it relies on user supplied input predicates. We show (in Section 5) how our iterative technique discovers invariants for (some of) the examples from [13] *without* any user supplied predicates.

The main contributions of our work are:

- A verification framework that combines symbolic execution and model checking in a novel way; we extend the basic framework presented in [23] with the ability to handle arrays symbolically and to prove partial-correctness specifications, that may be universally quantified. This results in a flexible and powerful tool that can be used for proving program correctness, in addition to test input generation and model checking.
- A new method for iterative invariant generation. The method handles uniformly different types of constraints (e.g. boolean and numeric constraints, arrays and objects) and it can be used in conjunction with more powerful approximation methods (e.g. widening [7, 9]).
- A series of (small) non-trivial Java examples showing the merits of our method; our method extends to other languages and model checkers.

```
// @ precondition: a != null;
void example(int[] a) {
1:  int i = 0;
2:  while (i < a.length) {
3:    a[i] = 0;
4:    i++;
    }
5:  assert a[0] == 0;
}
```

Fig. 1. Motivating example

Section 2 shows an example analysis in our framework. Section 3 gives background on symbolic execution and it describes our symbolic execution framework for Java programs. Section 4 gives our method for proving properties of Java programs using symbolic execution and invariant generation and Section 5 illustrates its application to the verification of several non-trivial Java programs. We give related work in Section 6 and conclude in Section 7.

2 Example

We illustrate our verification framework using the code shown in Figure 1. This method takes as a parameter an array of integers `a` and it sets all the elements of `a` to zero. This method has a `precondition` that its input is not null. The `assert` clause declares a partial correctness property that states that after the execution of the loop, the value of the first element in `a` is zero.

Using the loop invariant $i \geq 0$, our framework can be used to automatically check that there are no array bounds violations. This is a simple invariant that can be stated without much effort. In order to prove that there are no assertion violations, a more complex loop invariant is needed: $\neg(a[0] \neq 0 \wedge i > 0)$.

Constructing this loop invariant requires ingenuity. Our framework discovers this invariant by iterative approximation. It starts with $I_0 = \neg(a[0] \neq 0 \wedge i \geq a.length)$ which is the weakest possible invariant that is necessary to prove that the assertion is not violated. When checking this invariant to see if it is inductive we find a violation: if the formula $(i+1) \geq a.length \wedge a[0] \neq 0 \wedge 0 < i < a.length$ holds at the beginning of the loop, then I_0 does not hold at the end of the loop. At the next iteration, we strengthen I_0 using $a[0] \neq 0 \wedge 0 < i < a.length$ (i.e. we drop the *new constraint* $(i+1) \geq a.length$ that is due to the iterative computation in the loop body). This yields the formula: $\neg(a[0] \neq 0 \wedge i \geq a.length) \wedge \neg(a[0] \neq 0 \wedge 0 < i < a.length)$, which simplifies to the desired invariant.

Now suppose we want to verify an additional assertion, which states that, after the execution of the loop, every element in the array `a` is set to zero: $\forall\ int\ j : a[j] = 0$. This assertion is universally quantified; it refers to the quantified variable j as well to the program variables. We model it by introducing a symbolic constant j, which is a new variable that is not mentioned elsewhere

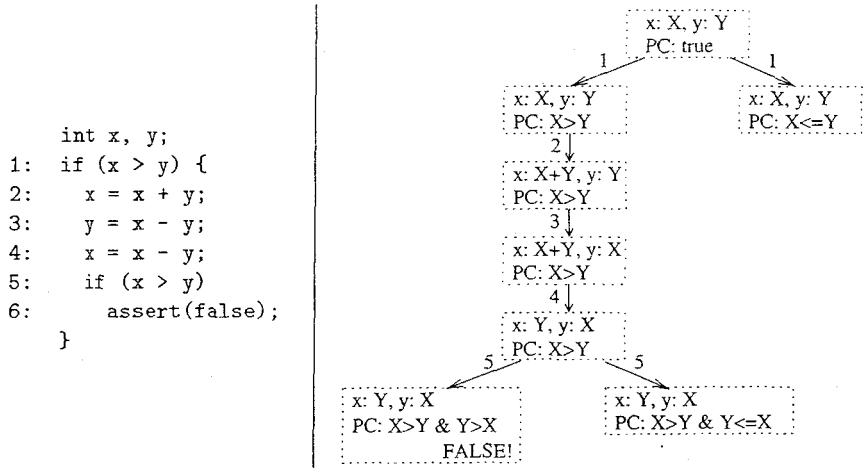

Fig. 2. Code that swaps two integers and the corresponding symbolic execution tree, where transitions are labeled with program control points

in the program and it is assigned a new, unconstrained symbolic value. Our symbolic execution framework automatically infers the loop invariant: $\neg(a[j] \neq 0 \wedge i \geq a.length \wedge 0 \leq j < a.length) \wedge \neg(a[j] \neq 0 \wedge j < i \wedge 0 \leq i, j < a.length)$.

Since the symbolic constant j represents some fixed unknown value, this invariant is valid for any value of j. This technique is crucial for checking programs that manipulate unbounded data, such as arrays [13].

3 Symbolic Execution in Java PathFinder

In this section we give some background on symbolic execution and we present the symbolic execution framework used for reasoning about Java programs.

3.1 Background: Symbolic execution

The main idea behind symbolic execution [24] is to use *symbolic values*, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the input symbolic values.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (PC) and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment in Figure 2, which swaps the values of integer variables x and y, when x is greater than y. Figure 2 also shows the corresponding symbolic execution tree. Initially, PC is *true* and x and y have symbolic values X and Y, respectively. At each branch point, PC is updated with assumptions about the inputs, in order to choose between alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and PC is updated accordingly. If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement (6) is unreachable.

3.2 Generalized Symbolic Execution

In [23] we describe an algorithm for generalizing traditional symbolic execution to support advanced constructs of modern programming languages, such as Java and C++. The algorithm handles dynamically allocated structures (e.g., lists and trees), method preconditions (e.g., acyclicity of lists), data (e.g., integers and strings) and concurrency. Partial correctness properties are given as assertions in the program and temporal specifications. We have since extended the work in [23] by adding support for symbolic execution of arrays and for checking quantified formulas.

3.3 Symbolic Execution Framework

Our symbolic execution framework automates test case generation and allows model checking concurrent programs that take inputs from unbounded domains with complex structure. To enable a model checker to perform symbolic execution, the original program is instrumented by doing a source to source translation that adds nondeterminism and support for manipulating formulas that represent path conditions. The model checker checks the instrumented program using its usual state space exploration techniques — essentially, the model checker explores the symbolic execution tree of the program. A *state* includes a heap configuration, a path condition on primitive fields, and thread scheduling. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure, such as the Omega library [27] for linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks.

Note that performing (forward) symbolic execution on programs with loops can explore infinite execution trees. This is why, for systematic state space exploration, the framework presented in [23] uses depth first search with iterative deepening or breadth first search. The framework can be used for test input generation and for finding counterexamples to safety properties. If there is an upper bound on the number of times each loop in the program may be executed, the framework can also be used for proving correctness, since the corresponding symbolic execution tree is finite.

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding execution trees are infinite. In order to

```
void example() {
  IntArrayStructure a =  new IntArrayStructure();
  Expression i = new IntegerConstant(0);

  while(Expression.pc._update_LT(i,a.length)) {
    a._set(i,new IntegerConstant(0));
    i = i._plus(new IntegerConstant(1));
  }
  assert Expression.pc._update_EQ(a._get(new IntegerConstant(0)),0);
}
```

Fig. 3. Instrumented code

prove the correctness of such programs, we have extended our framework with the ability of traversing the symbolic execution tree inductively rather than explicitly, using loop invariants (as presented in the next section).

3.4 Java PathFinder

Our framework uses the Java PathFinder(JPF) [30] model checker to analyze the instrumented programs. As a decision procedure, the framework uses a Java implementation of the Omega library.

JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). Since it is built on a JVM, it can handle all of the language features of Java, but in addition it also treats nondeterministic choice expressed in annotations of the program being analyzed — annotations are added to the programs through method calls to a special class `Verify`. These features (`Verify.choose_boolean()` and `Verify.choose(n)`) for adding nondeterminism are used to implement the updating of path conditions. JPF also supports a program annotation to forces the search to backtrack (`Verify.ignoreIf(condition)`) when a certain condition evaluates to true— this is used to stop the analysis of infeasible paths (when path conditions are found to be unsatisfiable).

3.5 Instrumentation

The interested reader is referred to [23] for a detailed description of how the code is instrumented for symbolic execution, here we will instead just highlight some key new features.

The main idea is to replace concrete types with corresponding "symbolic types" (i.e. library classes that we provide) and concrete operations with method calls that implement "equivalent" operations on symbolic types. As an illustration of the instrumentation, consider the code from Figure 1. Figure 3 gives part of the resulting code after instrumentation and Figure 4 gives part of the library classes that we provide. Classes `Expression` and `IntArrayStructure` support manipulation of symbolic integers and symbolic integer arrays, respectively.

```
class Expression { ...
 static PathCondition pc;
 Expression _plus(Expression e){
   ...} }

class PathCondition { ...
 Constraints c;
 boolean _update_LT(Expression l,
                    Expression r){
  boolean result;
  result=Verify.choose_boolean();
  if (result)
    c.add_constraint_LT(e1,e2);
  else
    c.add_constraint_GE(e1,e2);
  Verify.ignoreIf(!c.is_sat());
  return result;
  } }
```

```
class IntArrayStructure {
 Vector _v;
 Expression length;
 ...
 ArrayCell _new_ArrayCell(Expression idx) {
  for(int i=0;i<_v.size();i++) {
   ArrayCell cell=(ArrayCell)_v.elementAt(i);
   if(Expression.pc._update_EQ(cell.idx,idx))
    return cell;
  }
  ArrayCell t=new ArrayCell(length,idx,name);
  _v.add(t);
  return t;
 }
 public Expression _get(Expression idx) {
  assert(Expression.pc._update_GE(idx, 0)&&
       Expression.pc._update_LT(idx,length));
  ArrayCell cell = _new_ArrayCell(idx);
  return cell.elem;
 } }
```

Fig. 4. Library classes

The `static` field `Expression._pc` stores the (numeric) path condition. Method `_update_LT` makes a nondeterministic choice (i.e., a call to `choose_boolean`) to add to the path condition the constraint or the negation of the constraint its invocation expresses and returns the corresponding `boolean`. Method `is_sat` uses the Omega library to check if the path condition is infeasible (in which case, JPF will backtrack). Method `_plus` constructs a new `Expression` that represents the sum of its input parameters. `IntegerConstant` is a subclass of `Expression` and wraps concrete integer values.

To store the input array elements that are created as a result of a lazy initialization, we use a variable of class `Vector`, for each input array. The `_get` and `_set` methods use the elements in this vector to systematically initialize input array elements. When the execution accesses a symbolic array cell, the algorithm nondeterministically initializes it to a new cell or to a cell that was created during a prior cell initialization. The assertion checks in the `_get`/`_set` methods establish that there are no array out of bounds errors.

4 Proving Properties of Java Programs

In this section we present a Floyd-Hoare style method [14,18,20] for proving lightweight properties of Java programs. The method requires loop invariants and we present a novel iterative technique for discovering (some of) them automatically.

```
init;
while (C) {
  body;
}
assert P;
```

```
1:    init;
2:    assert I; /* base case */
3:    symbolic variables in B;
4:    assume I;
5:    if (C) {
6:      B;
        // oldPC
7:      assert I; /* induction step */
        // PC
      }
8:    else
9:      assert P;
```

Fig. 5. Single loop program (left) and instrumented program for proof (right)

4.1 Proving Properties using Symbolic Execution

For simplicity of presentation, we illustrate our methodology on a single-loop program such as the one in Figure 5 (left); multiple loops can be treated similarly, see e.g. [31]. The program consists of some (loop-free) initialization code, a loop with condition `C` and (loop-free) `body` and post condition `P`.

To verify the program, it suffices to find a loop invariant I, i.e. a formula that is true when entering the loop, re-entering the loop during its iteration and exiting the loop [18]. Moreover, I must be strong enough to produce verifiable results (hence a loop invariant *true* is, in general, not sufficient). In a symbolic execution framework, this amounts to checking the three assertions in the modified program in Figure 5 (right). Here, we replaced the `while` statement with an `if` statement; this is equivalent to placing a "cut" in the loop [18]. At this cut point, we consider all the variables that are modified in the loop body initialized to *new* symbolic values, and the path condition initialized to *true*. Note that a symbolic execution from this point on is representative of an arbitrary number of loop unrollings; the "input variables" at the cut point are the variables that are modified by the loop body and their new symbolic values represent all cases. Since the program loop has been cut, this symbolic execution will terminate and have a finite symbolic execution tree.

We check for three assertions :

- the assertion at line (4) is the *base case* of the inductive argument and checks that I holds when entering the loop
- the assertion at line (7) is the *induction step* and checks that, *assuming* I holds at the beginning of the loop, I also holds after the execution of the loop body (i.e. I is inductive)
- the assertion at line (9) checks that I is strong enough for the property to hold (i.e. $I \wedge \neg C \rightarrow P$)

If there are no assertion violations in the loop-free program of Figure 5 (right), then the program of Figure 5 (left) does not violate the property P. With this technique, we can verify properties of complex Java programs using the symbolic

execution framework presented in Section 3. However, the technique requires the generation and use of loop invariants.

4.2 Invariant Generation

The generation of loop invariants is an intricate problem that often requires a deep understanding of how these loops work.

We propose a novel technique for generating these loop invariants automatically. The technique works backward, starting from the property to be proved and it has three basic ingredients: iterative invariant strengthening, iterative approximation and refinement.

Iterative Invariant Strengthening Consider again the example in Figure 5. The check for the actual property (i.e. the assertion at line (9)) is used for defining the initial candidate invariant; the weakest possible choice is $I_0 = \neg(\neg C \wedge \neg P)$. If the base case fails for this candidate invariant, then the program is not verifiable (i.e. it has an assertion violation).

Checking the inductive step generates all the symbolic paths for the loop body. If for some of these paths, the invariant is not inductive, then it must be replaced by a stronger invariant. Assume $PC_1, PC_2, \ldots, PC_n$ are the path conditions for the paths on which the verification of the induction step fails. These path conditions characterize all the "inputs" to the loop body for which the check for the inductive step fails. The invariant is strengthened by replacing it with $I_1 = I_0 \wedge \neg PC_1 \wedge \neg PC_2 \wedge \ldots \wedge \neg PC_n$ and the base case and the inductive step are checked again.

If applied repeatedly, this process can introduce infinitely many new constraints, hence it can lead to an infinite sequence of *exact candidate invariants*[4] $I_1, I_2, \ldots$ We propose to use a simple, but powerful approximation technique to help termination.

Iterative Approximation At each step $k \geq 0$, we apply our approximation phase for the current candidate invariant I_k. We should first observe that symbolically executing the assumption and the body of the loop once (i.e. executing lines (4) through (6) in the code of Figure 5 (right)) will generate a *finite* number of symbolic execution paths, that contain a finite number of constraints; we call these constraints the *universe of constraints* U_k at step k. U_k contains the constraints from the current invariant together with the constraints generated by symbolically executing the loop body. New constraints (that are not in U_k) may get generated by the symbolic execution of the assertion at line (7).

[4] We distinguish between *exact* candidate invariants, that are generated during iterative invariant strengthening and *approximate* candidate invariants, that are generated during iterative approximation. If the base case fails for an exact invariant, then the program is not verifiable. But if the base case fails for an approximate invariant, this might indicate that the approximation was too coarse so it needs refinement.

Let PC be a path condition for some path in the loop body *after* checking and discovering a violation for the assertion at line (7), and let $oldPC$ be the path condition for the same path in the loop body, *before* checking the assertion. As we said, checking for the assertion itself can potentially add new constraints to the path condition (i.e. the set of constraints accumulated in $oldPC$ is a subset of the set of constraints in PC). In the approximation phase, instead of strengthening the invariant using PC, we use $oldPC$, which is *weaker* than PC (i.e. $PC \rightarrow oldPC$); this has the effect of obtaining a stronger invariant. In other words, our approximation consists of a strengthening step in which we drop all the newly generated constraints (e.g. constraints that are present in PC but not in $oldPC$, and hence not in U_k). The approximation phase generates a sequence of *approximate candidate invariants* $I_k^1, I_k^2, \ldots$; since there are only a finite number of constraints in U_k, this process is guaranteed to terminate, yielding an inductive invariant I_k^l, for some $l > 0$. I_k^l is a boolean combination of the constraints contained in U_k.

Refinement If the base case fails for an approximate invariant, this may be because the approximation is too strong. This means that the universe of constraints U_k is too coarse for proving the property and it needs to be refined. A simple refinement that we use is to consider U_{k+1} whenever the base case fails for an approximate invariant. This amounts to backtracking to the candidate invariant I_k, computing the next *exact candidate invariant* I_{k+1} and applying the approximation phase at the next iteration. Note that since the set of constraints in I_k is a subset of the set of constraints in I_{k+1}, we have that $U_k \subseteq U_{k+1}$, and hence U_{k+1} will yield finer approximation steps. We should also note that if the program has an error, it will be eventually caught when the proof of the base case will fail for an exact invariant.

Description of General Verification Method Now that we have seen the basic ingredients, here is how the general method for checking properties works. We use the check for the actual property to come up with the initial candidate invariant I_0. We then check the base case and the inductive step for this invariant.

- if both these checks yield no errors, then we are done, the result is that the property holds for the program and the current invariant is inductive
- if the inductive step fails, we apply *iterative approximation* to get a stronger invariant and we go back to checking the base case and the inductive step
- if the base case fails and the current candidate invariant is *exact*, then we are done, and the result is that the property does not hold for the program; if the base case fails and the current candidate invariant is *approximate*, we apply *refinement* and we check again the base case and the inductive step

If there is an error in the program, our method is guaranteed to terminate, reporting the error. However, if the program is correct with respect to the given property, this iterative method might not terminate (and the refinement might continue indefinitely).

```
void example() {
1:   IntArrayStructure a =  new IntArrayStructure();
2:   Expression i = new IntegerConstant(0);
3:   try {
4:    assert(I); /* base case */
5:    i = new SymbolicInteger();
6:    Expression j = new SymbolicInteger();
7:    Verify.ignoreIf(!I); /* assume I */
8:      if(Expression.pc._update_LT(i,a.length)) {
9:        a._set(i,new IntegerConstant(0));
10:       i = i._plus(new IntegerConstant(1));
11:       ... // oldPC = PC;
12:       assert(I); /* induction step */
13:     }
14:     else
15:       assert Expression.pc._update_EQ(a._get(new IntegerConstant(0)),0);
16:       // assert  Expression.pc._update_EQ(a._get(j),0);
17:   } catch (AssertionError e) {
18:       ... // print oldPC;
19:       ... // print PC;
    } }
```

Fig. 6. Motivating example - verification (excerpts)

4.3 Illustration

Consider again our motivating example program from Section 2. The program is instrumented to allow symbolic verification and inductive reasoning, as illustrated in Figure 6. Any assertion violation triggers an `AssertionError` exception, which is caught by the program (see lines (3) and (17)-(19) in the instrumented code). Variable `oldPC` stores the value of the path condition before the check of the inductive step; the value of `oldPC` is used in the approximation phase for invariant strengthening. Model checking the program using JPF prints all the path conditions `PC` (together with `oldPC`) for the assertion violations.

We first check the assertion at line (15) - which fails. The initial candidate invariant is then $I_0 = \neg(a[0] \neq 0 \wedge i \geq a.length)$. We now instrument this formula to enable symbolic execution and add it at lines (4), (7) and (12), then we model check the program and we find a counterexample for the following path condition(s):

$PC = (i+1) \geq a.length \wedge i > 0 \wedge a[0] \neq 0$;

$oldPC = i > 0 \wedge a[0] \neq 0$.

At this point we use iterative approximation, and we use $oldPC$ for strengthening the invariant (i.e. we drop the newly generated constraint $(i+1) \geq a.length$ from PC), yielding the new candidate invariant: $I_0^1 = I_0 \wedge \neg(i > 0 \wedge a[0] \neq 0)$. This invariant suffices to prove the property.

In order to check the additional assertion $\forall j : a[j] = 0$, we declare a new symbolic variable `j` (at line (6)) and we check for the assertion at line (16),

that is instrumented for symbolic execution. The initial candidate invariant is $I_0 = \neg(a[j] \neq 0 \wedge i \geq a.length \wedge 0 \leq j < a.length)$. Model checking the program using this additional invariant gives a counterexample for the following path condition(s):

$PC = (i+1) \geq a.length \wedge a[j] \neq 0 \wedge j < i \wedge 0 \leq i, j < a.length$;

$oldPC = a[j] \neq 0 \wedge j < i \wedge 0 \leq i, j < a.length$.

Using $oldPC$ for strengthening the invariant, we get $I_0^1 = I_0 \wedge \neg(a[j] \neq 0 \wedge j < i \wedge 0 \leq i, j < a.length$ which suffices to prove the property.

4.4 Discussion

We have presented a method that extends the framework presented in [23] with the ability of proving partial-correctness specifications. This yields a flexible framework for checking Java programs. The general methodology for using our framework is to first use it as a model checker, using depth first search with iterative deepening or breadth first search.

If no errors are found up to a certain depth, then there is some confidence that the program is correct (with respect to the given property), and a proof of correctness can be attempted using the method presented in this section. If an error is still present after the model checking phase, it will be found as a base case violation for an exact candidate invariant.

Our approximation consists of dropping newly generated constraints; a potentially more powerful, but more expensive, approximation would be that instead of dropping constraints, to replace them with an appropriate boolean combination of existing constraints from U_k. This has some similarities with the predicate abstraction techniques and we would like to investigate this further. Our technique can also be used in conjunction with other, more powerful methods [7–9, 32].

Our current system is not fully automated; although we discover all path conditions that lead to an assertion violation automatically, we combine the conditions by hand into a candidate invariant and add it back to the code to check if it is inductive. An implementation of these features is currently underway.

Traditionally, invariant generation has been performed using iterative forward and backward traversal, using different heuristics for terminating the iteration; e.g. convergence can be accelerated by using auxiliary invariants (i.e. already proved invariants or structural invariants obtained by static analysis) [3, 4, 16, 19, 25, 29, 31]. Abstract interpretation introduced the *widening* operator, which was used to compute fixpoints systematically [7–9]. Alternative methods [5] use constrained based techniques for numeric invariant generation.

Most of these methods use techniques that are *domain specific*. Our method for invariant generation uniformly treats different kinds of constraints. Our method could be viewed as an iterative-deepening search of a sufficient set of constraints that could express an invariant that is strong enough for verifying the property. Each step in this search is guaranteed to terminate, but deepening (refinement) may be non terminating.

```
// @ precondition: a != null && b != null && a.length == b.length;
int find(int [] a, boolean [] b) {
1:    int spot = a.length;
2:    for (int i = 0; i < a.length; i++) {
3:      if (spot == a.length && a[i] != 0)
4:        spot = i;
5:      b[i] = (a[i] != 0);
      }
6:    assert (spot == a.length || b[spot]);
7:    return spot;
}
```

Fig. 7. Method `find`

5 Experiments

This section shows the application of our framework to the verification of several non-trivial Java programs. We compare our work with the invariant generation method presented in [13]. We also show an example for which our method is not able to infer a loop invariant, in which case it can benefit from more powerful approximation techniques.

Method find Figure 7 shows an example adapted from [13]. Method `find` takes as parameters an array of integers `a` and an array of booleans `b`. The method returns the index of the first non-zero element of `a` if one exists and `a.length` otherwise. The method also sets the `i`-th element of `b` to true if the `i`-th element of `a` is nonzero, and to false otherwise. The preconditions of the method state that the arrays are not null and of the same length. The assertion states that the index to be returned (`spot`) is either `a.length` or `b` is true at that index.

To check that there are no assertion and array bounds violations, our framework infers the following invariant ($k = 0$, two approximation steps):

$\neg(i < 0) \wedge \neg(i \geq a.length \wedge 0 \leq spot < a.length \wedge \neg b[spot]) \wedge$
$\neg(0 \leq i < a.length \wedge spot = i \wedge a[i] = 0) \wedge$
$\neg(0 \leq i < a.length \wedge 0 \leq spot < i \wedge \neg b[spot]) \wedge$
$\neg(0 \leq i < a.length \wedge i < spot < a.length)$.

This invariant is sufficient to prove the property. As in [13] we checked an additional assertion, which states that, at the end of the method execution, every element of `b` before `spot` contains `false`: $\forall\ int\ j : 0 \leq j < spot \rightarrow \neg b[j]$.

To prove that this assertion holds, our framework generates the following additional invariant:

$\neg(i \geq a.length \wedge 0 \leq j < spot \wedge spot \leq a.length \wedge b[j]) \wedge$
$\neg(0 \leq i < a.length \wedge 0 \leq j < i \wedge spot = a.length \wedge b[j]) \wedge$
$\neg(0 \leq i < a.length \wedge 0 \leq j < spot \wedge spot = i \wedge b[j] \wedge a[i] \neq 0) \wedge$
$\neg(0 \leq i < a.length \wedge 0 \leq j < spot \wedge spot < i \wedge b[j] \wedge b[spot])$.

```
Node partition (Node l, int v) {
1:  Node curr = l;
2:  Node prev = null;
3:  Node newl = null;
4:  Node nextCurr;
5:  while(curr != null) {
6:    nextCurr = curr.next;
7:    if (curr.elem > v) {
8:      if (prev != null)
9:        prev.next = nextCurr;
10:     if (curr == l)
11:       l = nextCurr;
12:     curr.next = newl;
13:     assert curr != prev
14:     newl = curr;
      }
15:   else {
16:     prev = curr;
      }
17:   curr = nextCurr;
    }
18: return newl; }
```

```
void m(int n) {
1:  int x = 0;
2:  int y = 0;
3:  while (x < n) {
4:    x++;
5:    y++;
    }
6:  /* hint: x == y; */
7:  while (x != 0) {
8:    x--;
9:    y--;
    }
10: assert (y == 0);
}
```

Fig. 8. Method `partition` (left) and another example (right)

The method presented in [13] starts with a set of "interesting" predicates provided by the user and performs iterative forward *abstract computations* to compute a loop invariant as a combination of these predicates. For proving the first assertion in the example above, the method requires three predicates: $spot = a.length$, $b[spot]$ and $spot < i$, while for proving the second assertion, the method requires four additional predicates: $0 \leq j$, $j < i$, $j < spot$ and $b[j]$.

In contrast, our method does not require any user supplied predicates, although we should note that some of these predicates can be generated by several heuristic methods that are also described in [13]. We should also note that the invariants in [13] are more concise, as they are given in disjunctive normal form. Unlike [13], our method works backward starting from the property to be checked and it naturally discovers the necessary constraints over the program's variables, through symbolic execution and refinement. An interesting future research direction is to use the method presented in [13] in conjunction with ours: at each step k, instead of using approximation we could use the predicate abstraction based method, starting from the set of constraints U_k.

List Partition Figure 8 (left) shows a list partitioning example adapted again from [13]. Each list element is an instance of the class `Node`, and contains two fields: an integer `elem` and a reference `next` to the following node in the list. The method `partition` takes two arguments, a list `l` and an integer value `v`. It removes every node with value greater than `v` from `l` and returns a list containing

all those nodes. The assertion states that curr is not aliased with prev. Our framework checks that there are no assertion violations and it generates the following sequence of candidate invariants.

$I_0 = \neg(curr = prev \wedge curr \neq null \wedge curr.elem > v)$.

$I_0^1 = I_0 \wedge \neg(curr \neq prev \wedge curr \neq null \wedge prev \neq null \wedge curr.elem > v)$.

Approximate invariant I_0^1 is too strong (I_0^2 leads to a base case violation). The framework then backtracks and continues with the next exact invariant:

$I_1 = I_0 \wedge \neg(curr \neq prev \wedge curr \neq null \wedge prev \neq null \wedge curr.elem > v \wedge prev.elem > v \wedge prev = curr.next)$.

$I_1^1 = I_1 \wedge \neg(curr \neq prev \wedge curr \neq null \wedge prev \neq null \wedge curr.elem > v \wedge prev.elem > v \wedge prev \neq curr.next)$.

Approximate invariant I_1^1 is inductive. This example has shown that our framework can handle constraints on structured data. We also successfully applied our framework to the examples presented in [11], where we checked the absence of null pointer dereferences.

Pathological Example The iterative method for invariant generation presented in Section 4 might not terminate. For example, consider the code in Figure 8 (right)[5].

As our method works backward from the property, we first attempt to compute a loop invariant for the second loop. Our iterative refinement will not terminate for this loop. Considering increasing the number of exact strengthening steps does not help. Intuitively, the method does not converge because the constraint $x = y$ (and its negation) is "important" for achieving termination, but this constraint does not get discovered by repeated symbolic executions of the code in the loop body.

The programmer can provide additional helpful constraints by hand in the form of "hints", to boost the precision of the iterative approximation method. For example, the hint at line (6) in the code of Figure 8 (right) has the effect of nondeterministically adding the constraint (and its negation) to the current path condition, and hence these constraints are also added to the universe of constraints at each strengthening step. With this hint, we get the following loop invariant for the second loop ($k = 0$, two approximation steps):

$\neg(y \neq 0 \wedge x = 0) \wedge \neg(y \leq 0 \wedge x > 0) \wedge \neg(y > 0 \wedge x \neq y)$.

Using this invariant as the postcondition for the first loop, we then get the following loop invariant for the first loop, which suffices to prove the property: $\neg(x \geq n \wedge x \neq y) \wedge \neg(x < 0) \wedge \neg(x \geq 0 \wedge x < n \wedge x \neq y)$.

We should note that more powerful techniques such as linear equalities abstract domain [22] would work for this example. We would like to use our framework in conjunction with more powerful abstraction techniques (such as [22]) or with alternative dynamic methods for discovering loop invariants (e.g. the Daikon tool [12] could be used to provide useful "hints").

[5] Note that several other methods, such as the predicate abstraction with refinement as implemented in the SLAM tool [1] would also not terminate on this example.

6 Related work

Throughout the paper, we have discussed related work on invariant generation. Here we link our approach to software verification tools. King [24] developed EFFIGY, a system for symbolic execution of programs with a fixed number of integer variables. EFFIGY supported various program analyses (such as assertion based correctness checking) and is one of the earliest systems of its kind.

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [10] uses a theorem prover to verify partial correctness of classes annotated with JML specifications. ESC has been used to verify absence of such errors as null pointer dereferences, array bounds violations, and division by zero. However, tools like ESC rely heavily on specifications provided by the user and they could benefit from invariant generation techniques such as ours.

The Three-Valued-Logic Analyzer (TVLA) [28] is a static analysis system for verifying rich structural properties, such as preservation of a list structure in programs that perform list reversals via destructive updating of the input list. TVLA performs fixed point computations on shape graphs, which represent heap cells by shape nodes and sets of indistinguishable runtime locations by summary nodes. Our approximation technique has similarities to widening operations used in static analysis. We would like to explore this connection further.

The pointer assertion logic engine (PALE) [26] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers that do not add extra information. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. Shape analyses, such as TVLA and PALE, typically do not verify properties of programs that perform operations on numeric data values.

There has been a lot of recent interest in applying model checking to software. Java PathFinder [30] and VeriSoft [15] operate directly on a Java, respectively C program. Other projects, such as Bandera [6], translate Java programs into the input language of verification tools. Our work would extend such tools with the ability to prove partial-correctness specifications. The Composite Symbolic Library [33] uses symbolic forward fixed point operations to compute the reachable states of a program. It uses widening to help termination but can analyze programs that manipulate lists with only a fixed number of integer fields and it can only deal with closed systems.

The SLAM tool [1] focuses on checking sequential C code with static data, using well-engineered predicate abstraction and abstraction refinement tools. It does not handle dynamically allocated data structures. Symbolic execution is used to map abstract counterexamples on concrete executions and to refine the abstraction, by adding new predicates discovered during symbolic execution. We should note that tools like SLAM perform abstraction on each program statement, whereas our method performs approximation (which can be seen as a form of abstraction) only when necessary, at loop headers. This indicates that our method is potentially cheaper in terms of the number of predicates (i.e. constraints) required. Of course, further experimentation is necessary to support

this claim. There are many similarities between predicate abstraction and our iterative approximation method and we would like to compare the two methods in terms of relative completeness (as in [2, 9]).

7 Conclusion

We presented a novel framework based on symbolic execution for the verification of software. The framework uses annotations in the form of method specifications and loop invariants. We presented a novel iterative technique for discovering these loop invariants automatically. The technique works backward from the property to be checked and it systematically applies approximation to achieve termination. The technique handles uniformly both numeric constraints and constraints on structured data and it allows for checking universally quantified formulas. We illustrated the applicability of our framework to the verification of several non-trivial Java programs. Although we made our presentation in the context of Java programs, JPF, and the Omega library, our framework can be instantiated with other languages, model checkers and decision procedures.

In the future, we plan to investigate the application of widening and other more powerful abstraction techniques in conjunction with our method for invariant generation. We also plan to extend our framework to handle multithreading and richer properties. We would also like to integrate different (semi) decision procedures and constraint solvers that will allow us to handle floats and non-linear constraints. We believe that our framework presents a promising flexible approach for the analysis of software. How well it scales to real applications remains to be seen.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI*, pages 203–213, 2001.
2. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. TACAS*, 2002.
3. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Proc. CAV*, 1996.
4. N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *FMSD*, 16:227–270, 2000.
5. M. Colon, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, 2003.
6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. ICSE'00.*
7. P. Cousot and R. Cousot. On abstraction in software verification. In *Proc. CAV'02.*
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th POPL*, 1978.
9. G. Delzanno and A. Podelski. Widen, narrow and relax. Technical report.
10. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

11. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. SAS*, 2000.
12. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proc. ICSE*. ACM, 2000.
13. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, 2002.
14. R. W. Floyd. Assigning meanings to programs. In *Proc. Symposia in Applied Mathematics 19*, pages 19–32, 1967.
15. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL*, pages 174–186, Paris, France, Jan. 1997.
16. S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Proc. 8th CAV*, pages 196–207, 1996.
17. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th CAV*, pages 72–83, 1997.
18. S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Comput. Surv.*, 8(3):331–353, 1976.
19. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proc. FME*, pages 662–681, 1996.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
21. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* 2003.
22. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.
23. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, 2003.
24. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
25. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* 1992.
26. A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. PLDI*, Snowbird, UT, June 2001.
27. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), Aug. 1992.
28. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, January 1998.
29. A. Tiwari, H. Rues, H. Saidi, and N. Shankar. A technique for invariant generation. In *Proc. TACAS*, 2001.
30. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE*, Grenoble, France, 2000.
31. B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, 1974.
32. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. CAV*, pages 88–97, 1998.
33. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proc. SAS*, 2002.