Improving Coverage of Test Cases Generated by Symbolic PathFinder for Programs with Loops

Rody Kersten
Radboud University
Nijmegen, The Netherlands
r.kersten@cs.ru.nl

Suzette Person
NASA Langley RC
Hampton, Virginia, USA
suzette.person@nasa.gov

Neha Rungta
NASA Ames RC
Mofett Field, California, USA
neha.s.rungta@nasa.gov

Oksana Tkachuk
NASA Ames RC
Mofett Field, California, USA
oksana.tkachuk@nasa.gov

ABSTRACT

Symbolic execution is a program analysis technique that is used for many purposes, one of which is test case generation. For loop-free programs, this generates a test set that achieves path coverage. Program loops, however, imply exponential growth of the number of paths in the best case and non-termination in the worst case. In practice, the number of loop unwindings needs to be bounded for analysis.

We consider symbolic execution in the context of the tool Symbolic Pathfinder. This tool extends the Java Pathfinder model-checker and relies on its bounded state-space exploration for termination. We present an implementation of k-bounded loop unwinding, which increases the amount of user-control over the symbolic execution of loops.

Bounded unwinding can be viewed as a naive way to prune paths through loops. When using symbolic execution for test case generation, branch coverage will likely be lost when paths are naively pruned. In order to improve coverage of branches within a loop body, we present a technique that semi-automatically concretizes variables used in a loop. The basic technique is limited and we therefore present annotations to manually steer symbolic execution towards certain branches, as well as ideas on how the technique can be extended to be more widely applicable.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, testing tools*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Verification, Algorithms

Keywords

Symbolic execution, branch coverage, test case generation

1. INTRODUCTION

Testing is the most widely used technique for detecting faults in software. Software companies often dedicate over 50% of development time to testing. For safety-critical applications, this number is even larger. Composing an extensive set of test inputs is a complicated task, as the test designer must achieve some form of *coverage*. For instance, statement coverage requires that all statements in the program have been executed at least once.

Symbolic execution is a well-known technique from program analysis, which can be used for test case generation. In symbolic execution, a program is executed with symbols in place of concrete input. A *path-condition* is maintained, i.e., updated on each branch, that indicates the constraint under which this path is followed. Effectively, this means that if the generated constraints lie within the set of decidable theories, symbolic execution enumerates all paths through the software and the generated test cases provide full path-coverage. However, in programs with loops, any extra iteration of a loop introduces a new path, introducing exponential growth in the number of paths. Moreover, in loops that depend on input values, the number of paths may be infinite (81.8% of loops in the applications studied in survey paper [13] by Xiao et al. are input-dependent). Therefore, in practice, symbolic execution has to be bounded. Since in general, it is impossible to know a priori how many iterations are needed to enter certain branches, this means that likely, any notion of coverage is lost.

We consider symbolic execution in the context of the tool Symbolic Pathfinder (SPF) [8], which combines the model-checker Java Pathfinder [5] with symbolic execution and constraint solving to, among other objectives, generate test cases. SPF currently implements bounding of the search-space explored by the model-checker rather than bounded unwinding of loops. This means that the number of unwindings of loops is affected by the structure and complexity of the surrounding code. It is therefore hard to predict the number of unwindings for a particular loop, especially if other loops are present. We present the implementation of a more flexible and intuitive bounding mechanism for loops: k-bounded unwinding, making it possible to unwind the same number of iterations for each loop. Additionally, we present an annotation to specify k-bounds that are loop-specific.

Both types of bounding (loop bounding and search space bounding) may cause important paths through a program to be missed by SPF. When used to generate a set of test cases, this means that the test set will not cover all branches in the loop body. Paths are pruned by naively dropping all with more than k iterations, which can make it very complicated to achieve high coverage. We strive to improve the *object branch coverage* of the set of test cases generated by SPF. A set of test inputs provides object branch coverage if running the program for those test inputs executes every branch in the bytecode level control-flow graph.

Since in the bytecode level control-flow graph (CFG), evaluation of a condition such as $b1 \wedge b2$ amounts to two CFG nodes, as opposed to a single node in the source code level CFG, object

branch coverage implies branch coverage. Object branch coverage is thus a more rigorous coverage metric than source-level branch coverage.

We present an annotation which can be used to concretize symbolic variables. This can be used to fix symbolic variables to a set of concrete values that cover all branches in the loop body. Furthermore, we present a technique which can infer these cases for loops that are independent of context. The approach is limited, but represents a small step in improving the object branch coverage of the generated test set. Our contribution is thus threefold:

1. An implementation of k-bounded unwinding of loops in SPF.
2. A separate concretization technique, using annotations to concretize variables upon loop entry.
3. An experimental method to semi-automatically infer these annotations, based on out-of-context symbolic execution of the loop body.

The source code of our implementation of k-bounding and concretization using annotations can be found at `http://www.cs.ru.nl/R.Kersten/jpf-symbc-loops.tar.gz`.

1.1 Related Work

A good survey on symbolic execution for software testing is given in [2]. Several extensions to classical symbolic execution and state-of-the-art tools are discussed.

A survey on loop problems for *dynamic* symbolic execution (DSE) is given in [13]. DSE executes the program using concrete random inputs and collects the path condition on the side. An interesting result is that 81.8% of loops in the studied applications are dependent on input, thus possibly non-terminating. The most common way to deal with this type of loops is bounded iteration, solving the termination problem at the cost of completeness of the generated test set. Search-guiding heuristics can be used to guide symbolic execution to certain "interesting" paths, making the pruning less naive. A more complex approach is to create loop summaries: a set of formulas based on loop invariants and induction variables that summarizes the effects of the loop. This is a complex task which is infeasible for many loops. In fact, the state-of-the-art loop summarization algorithm presented in [4] can only summarize 6 out of 19 input-dependent loops in their experiment.

Single-path symbolic execution is a variation of DSE, in which a set of executions which follow the same control-flow path is considered. It is extended with a mechanism for loops in [11]. Iteration counters named *trip count variables* are introduced that can be linked to a known input grammar. It is shown that this is a powerful tool for finding problems such as buffer overflow vulnerabilities.

Verification of program properties using SPF is discussed in [9]. Loops are handled using invariants. Verification of a post-condition of a loop can be simplified to verification of the loop invariant before executing the loop (base case), after a generic iteration using symbolic execution (inductive case), plus implication of the post-condition from the invariant.

Gladisch describes a method to generate a test set with full feasible branch coverage, using the theorem prover KeY in [3]. It requires that strong preconditions, postconditions and loop invariants are supplied and leverages the theorem prover to replace symbolic execution of a loop by the application of a loop invariant rule. Several types of preconditions are formed for loops, which guarantee the execution of all branches in and after the loop.

Trtík presents a technique for handling loops in symbolic execution in [12]. He introduces *path counters* with update paths (increment by one) and reset paths (set to zero). Symbolic values of program variables can then be expressed in terms of these counters. This theoretically tackles the problem in part, but more complex loops quickly result in non-linear constraints which are expensive to solve or even undecidable.

2. BOUNDING LOOPS IN SPF

In this section we present our implementation of k-bounded unwinding of loops in SPF Consider the method in Listing 1. As i is assigned a symbolic value, symbolic execution of this program iterates over the loop infinitely many times. In practice, symbolic execution is bounded. SPF currenyly does not implement bounding itself, but instead relies on the bounded state-space exploration implemented in Java Pathfinder. This means that the number of unwindings of loops is affected by the structure and complexity of the surrounding code.

```
1 boolean m(int i) {
2   boolean b = false;
3   while (i>0) {
4     if (i==10)
5       b = true;
6     i--;
7   }
8   return b;
9 }
```

Listing 1: Example program with a loop.

To cover all branches, it is desired to unwind the loop in Listing 1 at least 10 times. Say we have a simple `main` method that initializes the object and then calls this method on it. If we set the depth-limit on the number of explored states to 4, the loop will be unwound only once, because of the other states on the path related to calling the method from the `main` function. The depth bound is based on the number of `ChoiceGenerator` objects that are encountered by Java Pathfinder. It cannot distinguish between choices related to a loop or other points of non-determinism. It gets harder to estimate the number of unwindings when there are other choice points on the path. Say, if there was a single if-statement after the loop, a depth-limit of 5 would be needed to unwind the loop once. Moreover, the number of choice points may differ between paths. Therefore, a different number of iterations might be unwound for the same loop in different paths. This makes it very hard to control the number of unwindings that will actually be done for a given loop.

2.1 K-Bounded Unwinding

We implemented k-bounded unwinding in SPF, in a listener called the `KBoundedSearchListener`. When this class is initialized, the CFG is built and the dominance set is calculated, in order to detect headers and back-edges of loops (a loop header is the single point of entry into the loop). A node n_1 dominates a node n_2 if all paths from the entry node to n_2 go through n_1. If an edge exists in the CFG from n to h and h dominates n, then the edge is the back-edge of a loop with header h. This loop detection algorithm is implemented in the `LoopFinder` class. Headers are stored in objects of the `Location` class, which combines a method name and an instruction position. The instructions of the choice

points following the loop headers are also stored, because that is where the actual branching occurs (loop headers typically consist of a load instruction).

The listener then counts the number of unwindings of each loop, using a stack, by listening for registered `ChoiceGenerator` objects. These are objects that JAVA PATHFINDER uses to navigate over decisions, where paths branch. When the k-bound is reached for a certain loop, the remaining paths through this loop are pruned by setting the next `ChoiceGenerator` to done.

Bounded unwinding is activated by adding the listener to the JAVA PATHFINDER configuration and setting the configuration option `kbound=K`, where `K` is the maximal number of unwindings. The implementation currently is limited to intra-procedural analysis (only loops within the analysed method itself are detected and bound, not those in called methods). An inter-procedural version will be implemented in the near future.

2.2 Specifying Loop-Specific Bounds

In some cases, one might want a certain loop to be unwound more than others, or maybe it is clear that unwinding it only once is enough. For those cases we have added an annotation to express loop-specific bounds. It is added to the header of a *Java* method and has the following syntax:

$$@KBound(k = \{\text{“}N_1 : b_1\text{”}, \ldots, \text{“}N_n : b_n\text{”}\})$$

Where each N_i is a loop identifier, determined by the order of loop headers from the top of the method in its source code, and each b_i is an integer bound.

3. CONCRETIZING LOOP VARIABLES

Typically, when symbolically executing a loop, up to a fixed number of k iterations are unwound and the path condition includes propositions expressing the number of unrolled iterations. For example $i > 0 \wedge i - 1 > 0 \wedge i - 2 \leq 0$ signifies two unrolled iterations for the example in Listing 1. As the number of iterations of loops is potentially infinite, a selection of paths through loops needs to be pruned. Unwinding of loops up to a given bound can often be ineffective in achieveing our testing goals, e.g. object branch coverage.

As we are considering symbolic execution in the context of test case generation, we are interested in obtaining test sets with better coverage. We therefore propose to prune paths through loops based on object branch coverage, by concretizing variables that are used in a loop to values that ensure coverage of all branches within its body. For instance, for the loop in Listing 1, concrete values 0, 1 and 10 might be used for i. We present an annotation for concretization in this section. A method for inferring usable concrete values is described in Section 4. The annotation is added to the method-header and has the following syntax:

$$@UseModels(models = \{C_1, \ldots, C_k\})$$

Where each C_x represents a concretization string:

$$C_x := \text{“}N_x.[v_1^x, \ldots, v_j^x] \rightarrow [m_1^x, \ldots, m_j^x]\text{”}$$

Where N_x is the identifier of a loop (determined by the order of loop headers from the top of the method in its source code), v_a^x is a program variable to be concretized and m_a^x is the model to concretize it to. In each of the k concretization strings, several variables may be concretized that might differ from the other concretization strings. When only a single variable and model combination is used, brackets may be omitted. As an example, to concretize i to 20 in the first loop of a method one can use the following annotation:

$$@UseModels(models = \{\text{”}1.i \rightarrow 20\text{”}\})$$

When concretizing, an equality between the model-value and the symbolic value of the program variable *upon entry to the loop* is added to the path condition. E.g., if the value of a program variable i is $i+3$ before the loop and there is an annotation that i should be concretized to 20, then $20 = i + 3$ is added to the path condition.

By concretizing the symbolic variables to these models, we prune all other paths, making the search-space finite. Concretization can thus replace other bounding methods. Note, however, that all iterations corresponding to the bound will need to be unrolled. For instance, if for our runnning example, a model m is found, the loop needs to be unrolled m times.

When variables are concretized for a loop, they are concretized for the program as a whole. Thus, when symbolically executing nested loops, concretization for an inner loop also affects the outer loop (and the rest of the program). Thus, care must be taken not to add conflicting annotations.

4. OUT-OF-CONTEXT SYMBOLIC EXECUTION OF THE LOOP BODY

In this section we present a technique that can infer variable values to concretize to for branch coverage of the loop body. It is inspired by [9], in which a loop body is symbolically executed with fresh symbols in order to prove a loop invariant. This technique is not yet automated. It consists of the following steps:

1. Symbolically execute the loop body out-of-context, i.e., with fresh symbols.
2. Solve the generated path conditions to obtain a *model* for each of them.
3. Concretize the values of the variables by adding the concrete values to the path condition (e.g., for models $i = 0$ and $i = 1$ we add $i = 0 \vee i = 1$).

Thanks to using *fresh* symbols for program variables, the search will not be biased to their symbolic values before entering the loop. The result of step 1 is a set of path conditions, capturing all behaviors one iteration of the loop can exhibit. Models for these path conditions can then be found using off-the-shelf constraint solvers.

Only variables that are used in the loop body or loop guard should be concretized. Otherwise, symbolic execution will settle on a limited set of paths through the entire program. These are also the only variables that need to be fresh for the out-of-context symbolic execution. Variables for which the model is not constrained by the path condition will also not be concretized, as these do not influence the flow of control in the loop.

4.1 Example

When symbolically executing the method in Listing 1 with k-bounded unwinding and $k = 2$, the following 3 path conditions are generated:

$$i \leq 0 \tag{1a}$$
$$i > 0 \wedge i \neq 10 \wedge i - 1 \leq 0 \tag{1b}$$
$$i > 0 \wedge i \neq 10 \wedge i - 1 > 0 \wedge i - 1 \neq 10 \wedge i - 2 \leq 0 \tag{1c}$$

Using the YICES solver, we get models $i = 0$, $i = 1$ and $i = 2$. The branch where b is assigned a value of *true* is missed. Let us now take the loop body out of context. A new method containing the extracted body is shown in Listing 2. The `while` has been replaced by an `if`, because we want the resulting models to satisfy the loop guard (except for the model that we also need in which the loop is not entered).

```
1 void outofcontext(int i, boolean b) {
2   if (i>0) {
3     if (i==10)
4       b = true;
5     i--;
6   }
7 }
```

Listing 2: The loop body from Listing 1, taken out of context.

Symbolic execution of this extracted loop body results in the following path conditions:

$$i \leq 0 \tag{2a}$$
$$i > 0 \wedge i \neq 10 \tag{2b}$$
$$i > 0 \wedge i = 10 \tag{2c}$$

Using the YICES solver, we get models $i = 0$, $i = 9$ and $i = 10$. Because there are no statements before the loop, we can simply use these symbols in the path condition as-is (no mapping, as explained in Section 3, is needed).

The paths through the loop can now be pruned in a more informed manner which enables object branch coverage by adding the models to the path condition. One can think of this as adding the following assumption before the loop:

```
assume (i==0 || i==9 || i==10);
```

Note that using the path conditions instead of the models would not prune the paths. A k-bound with $k \geq 10$ would still be needed to cover all branches.

4.2 Limitations

The concretization approach to handle loops has two major limitations. We discuss these here, including ideas on how we intend to address them in the future. Given these limitations, it is recommended to use the loop concretization technique for test case generation in combination with a coverage checker. Such a tool can check if the test set achieves object-branch coverage and point to branches that are missed. The user can then go back and add annotations to direct SPF to improve the generated test set.

Context-dependence. Out-of-context symbolic execution finds models for the out-of-context loop body. In cases such as the running example of this paper, adding these models to the path condition achieves object branch coverage, because the execution of the loop does not depend on this context. However, when the execution of the loop-body is dependent on the context, the models may be infeasible and the search will back-track. Consider, for instance, the loop in Listing 3. This method is taken from the Java prototype of the Airborne Coordinated Conflict Resolution and Detection (ACCoRD) framework developed and maintained by the NASA Langley formal methods group[1]. This is a framework for formal specification and verification of state-based airspace separation assurance algorithms. This specific method estimates the change of vertical speed from a sequence of velocity vectors stored in the containing object. The `numPtsVsRateCalc` parameter specifies the number of data points used in the average and the method returns the vertical acceleration. The sign of the return value indicates the direction of the acceleration.

```
1 public double avgVsRate(int numPtsVsRateCalc) {
2   int n = size();
3   if (numPtsVsRateCalc < 2) numPtsVsRateCalc = 2;
4   int numPts = Math.min(numPtsVsRateCalc,n);
5   double vsLast = 0;
6   double tmLast = 0;
7   double vsRateSum = 0.0;
8   for (int i=n-1; i>n-numPts-1 && i>=0; i--) {
9     StateVector svt = get(i);
10    double vs = svt.v().vs();
11    double tmTr = time(i);
12    if (i < n-1) {
13      double vsRate = (vs-vsLast)/(tmTr-tmLast);
14      vsRateSum = vsRateSum + vsRate;
15    }
16    vsLast = vs;
17    tmLast = tmTr;
18  }
19  if (numPts < 2) return 0;
20  else return vsRateSum/(numPts-1);
21 }
```

Listing 3: Loop for which its execution is dependent on its context, taken from the ACCoRD conflict resolution and detection framework.

If we analyze the body of the loop at lines 8-18 out-of-context, we get the following models (each row corresponds to a path through the loop body; other variables are omitted because they do not influence the control-flow in the loop and are therefore not concretized):

i	n	$numPts$
47	89	97
0	0	0
-24	-43	89
-77	86	92

The problem with these models is that the value of $numPts$ is defined as the minimum of $numPtsVsRateCalc$ and n on Line 4, but none of these models except for one satisfy the consequential constraint $numPts \leq n$. Furthermore, the path condition upon entry to the loop will contain a constraint $i = n-1$, as this is what i is initialized to on Line 8. This constraint is also not satisfied by any of the models. Therefore, when we add these constraints to the path condition, none of the paths through the loop are feasible.

We intend to create a refinement loop, which iteratively refines the constraint to satisfy with a model. The constraint is first set to the path condition of the path we are working on. The

[1] `http://shemesh.larc.nasa.gov/people/cam/ACCoRD/`

model that is found by the solver is then checked against the path conditions of the paths leading to the loop. If all of these conflict with the model, the constraint is strengthened with the conflicting sub-constraint of the path conditions. Complexity lies in finding this conflicting sub-constraint. Futhermore, we will introduce an annotation to specify whether or not loop variable concretization should be used.

Iteration-count dependence. Consider the example in Listing 4. There is only a single path through the loop body. Its path condition is $i > 0$ and a model is $i = 1$. When considering only this path, the path where the method does "something" (Line 8) is missed.

```
1 void m(int i) {
2   int j = 0;
3   while (i>0) {
4     j++;
5     i--;
6   }
7   if (j==20) {
8     //do something
9   }
10 }
```

Listing 4: Loop which shows dependence on iteration count.

In this case, the problem can be solved by using a manual annotation that states that i should be concretized to 20. There may also be branching in the loop that only occurs for a particular iteration, the conditional on Line 12 of Listing 3 is an example of this. The condition used in this case is true for any path, except for the first one. Such a case may be solved by setting a k-bound that is high enough.

The general case, where a branch may occur after n iterations, is equivalent to the halting problem. However, this does not mean that certain cases may not be tackled. An idea to improve this, is to add the iteration count as a variable to symbolic execution or detect induction variables, as is done e.g. in [12] and [4]. The symbolic value of variables can then be expressed using the number of iterations of each loop. The technique works with nested loops. In that case, the technique should be applied for the innermost loop first. The concrete values can then be used when the technique is applied to the outer loop(s).

5. CONCLUSIONS

We have presented a series of improvements to the loop-handling capabilities of Symbolic Pathfinder:

- An implementation of k-bounded unwinding of loops.
- Novel annotations to concretize variables, directing symbolic execution towards otherwise missed branches.
- A technique to infer concrete values for concretization.

The inference method has two major limitations: 1. when loops are context-dependent, and 2. when program variables are dependent on the number of loop iterations. We suggest some extension ideas to address these limitations.

The problem of pruning paths through loops is a notoriously hard one. A large body of literature on the topic exists and no proposed solution is complete. Our work represents a series of small steps towards better treatment of loops.

Future Work We will develop the ideas discussed in Section 4.2 of this paper and implement them in SPF. Furthermore, in [7, 6, 10, 1], an extension to SPF is discussed that compares software versions and generates test cases for paths that are impacted by changes only. This *incremental* analysis implies a significant reduction in the number of generated test cases. Extensions of our method that are specific for incremental analysis can potentially reduce this number even further and improve object branch coverage.

6. REFERENCES

[1] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *Model Checking Software*, LNCS 7976, pages 99–116. Springer, 2013.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE'11*, pages 1066–1071. ACM, 2011.

[3] C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *SEFM '08*, pages 159–168, Nov 2008.

[4] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA'11*, pages 23–33. ACM, 2011.

[5] K. Havelund and T. Pressburger. Model checking java programs using Java Pathfinder. *Int. Journal on Softw. Tools for Tech. Transfer*, 2(4):366–381, 2000.

[6] E. Mercer, S. Person, and N. Rungta. Computing and visualizing the impact of change with Java PathFinder extensions. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov 2012.

[7] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE '08*, pages 226–237. ACM, 2008.

[8] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic execution of Java bytecode. In *ASE '10*, pages 179–180. ACM, 2010.

[9] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Model Checking Software*, LNCS 2989, pages 164–181. Springer, 2004.

[10] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM '12*, pages 109–118, Sept 2012.

[11] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA '09*, pages 225–236. ACM, 2009.

[12] M. Trtík. *Symbolic Execution and Program Loops*. PhD thesis, Faculty of Informatics, Masaryk University, 2013.

[13] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *ASE '13*, pages 246–256, 2013.