Symbolic PathFinder v7

Kasper S. Luckow
Department of Computer Science
Aalborg University, Denmark
luckow@cs.aau.dk

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley, NASA Ames
Moffett Field, CA, USA
corina.s.pasareanu@nasa.gov

ABSTRACT

We describe Symbolic PathFinder v7 in terms of its updated design addressing the changes of Java PathFinder v7 and of its new optimization when computing path conditions. Furthermore, we describe the Symbolic Execution Tree Extension; a newly added feature that allows for outputting the symbolic execution tree that characterizes the execution paths covered during symbolic execution. The new extension can be tailored to the needs of subsequent analyses/processing facilities, and we demonstrate this by presenting SPF-Visualizer, which is a tool for customizable visualization of the symbolic execution tree.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; D.2.5 [**Testing and Debugging**]: Symbolic Execution; D.2.5 [**Testing and Debugging**]: Debugging Aids

General Terms

Verification, Design, Experimentation

Keywords

Java PathFinder, Symbolic PathFinder, Debugging, Software Engineering

1. INTRODUCTION

Symbolic execution [5] is a well known program analysis technique that executes a program on symbolic inputs instead of concrete data and computes the effects of the program as symbolic expressions in terms of these inputs. These symbolic inputs account for all the possible concrete inputs. The analysis explores the execution tree of a program, and for each explored path, it maintains a *path condition*, i.e. a conjunction of constraints on the inputs that must be satisfied to follow that path. Solving these constraints gives information about path feasibility and produces solutions that can be used as test inputs guaranteed to achieve high coverage through the code. The analysis has many applications, most notably automated test input generation and systematic error detection. Symbolic PathFinder (SPF) is a tool for performing symbolic execution of Java Bytecode. SPF handles inputs and operations on booleans, integers, reals, and complex data structures, as well as multi-threading, via integration with the Java PathFinder (JPF) model checker. The tool is open-source[1], with a large user base from NASA, academia, and industry (most notably Fujitsu).

This paper concentrates on the newly released v7 of SPF that introduces fundamental changes and new features. Most notably it has been redesigned to address JPF-core v7 including a migration to Java 7, and has furthermore been optimized for mitigating state space exploration. Accompanying SPF v7 is the Symbolic Execution Tree Extension that allows for easy post-processing of the symbolic execution tree structure. We incorporate in SPF v7 an application of the Symbolic Execution Tree Extension called SPF-Visualizer, which produces a visualization of the symbolic execution trees for the symbolically executed methods.

2. SPF V7

SPF is part of the Java PathFinder verification toolset [4]. This toolset includes JPF-core, which is an explicit-state model checker, and several extension projects, one of them being SPF. The model checker consists of an extensible Java Virtual Machine (JVM), state storage, and backtracking capabilities, different search strategies, and listeners for monitoring the search through a program's state-space. JPF-core executes the program concretely based on the standard semantics of Java. In contrast, SPF replaces the concrete execution semantics of JPF-core with a non-standard interpretation to enable symbolic execution. SPF relies on the JPF-core framework to systematically explore the symbolic execution paths, as well as different thread interleavings. To limit the possibly infinite search space that results from symbolically executing programs with loops or recursion, a user-specified depth is provided. To solve the path conditions, SPF uses several off-the-shelf solvers such as CHOCO [3], CORAL [8], and CVC3 [1].

The symbolic execution of branching conditions involves creating a non-deterministic choice in JPF's search and adding the condition, or its negation, to the path conditions. This is achieved by means of a `PCChoiceGenerator` that branches the execution inside JPF. A path condition is associated with each choice generated by the `PCChoiceGenerator`.

SPF v7 consists of 43 KLOC distributed on 270 source code files excluding test cases and examples. The size of SPF v6 in terms of KLOC is approximately the same, but distributed among 239 source code files. This difference is attributed that several parts of SPF have been refactored in v7. The most significant updates to the code are related to addressing the new `Instruction` interface which resulted in different ways of accessing the system state and associated path condition choices via the $ThreadInfo$ argument of the *execute* methods. Other notable code updates are related to the changed interfaces of the `VMListener` and `InstructionFactory`. Finally, the implementation of the lazy initialization algorithm for handling input data structures was updated significantly to reflect the new memory model in JFP-core. For SPF v7, we have also carefully revised the handling of multi-threading in the context of symbolic execution for GETFIELD and GETSTATIC. During our development, we discovered and reported several errors related to handling of multi-threading in JPF-core; these issues have been fixed. Work in progress includes updating the symbolic string analysis and careful testing of the code.

[1]Project website for SPF: `http://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc`

2.1 Path Condition Choice Optimization

We describe here an optimization that we made with respect to computing path conditions. We first explored this optimization in the context of using partial evaluation for compositional symbolic execution [7].

In the previous version of SPF, an instance of `PCChoiceGenerator` is created whenever a branching condition is executed, and the decision predicate is symbolic. For the branching instructions e.g. IFEQ and IFGT, the `PCChoiceGenerator` introduces two choices corresponding to the two branches and adds to the path condition the decision predicate and its negation, respectively. For long and floating point data type comparison instructions e.g. LCMP and FCMPL, three choices are introduced corresponding to the three different outcomes of the comparison. An excerpt of the original algorithm is shown in Listing 1 for the IFGT Java Bytecode.

```
1 if (!ti.isFirstStepInsn()) {
2   cg = new PCChoiceGenerator(2);
3   ...
4   sysState.setNextChoiceGenerator(cg);
5   return this;
6 } else {
7   cg = sysState.getChoiceGenerator();
8   conditionValue = (Integer)cg.getNextChoice() == 0
        ? false : true;
9 }
10 ...
11 if (conditionValue) {
12   pc._addDet(Comparator.GT, sym_v, 0);
13   if(!pc.solve())
14     sysState.setIgnored(true);
15   else
16     cg.setCurrentPC(pc);
17   return getTarget();
18 } else {
19   pc._addDet(Comparator.LE, sym_v, 0);
20   if(!pc.solve())
21     sysState.setIgnored(true);
22   else
23     cg.setCurrentPC(pc);
24   return getNext(ti);
25 }
```

Listing 1: Original algorithm for introducing path condition choices.

The condition in line 1 will evaluate to true upon the first execution of the instruction, thus adding a path condition choice to the current system state. This marks the end of the transition. Subsequently, it returns itself which will enable JPF to execute the instruction again; this time the condition in line 1 evaluates to false because a new transition has been created, and thus a choice on the condition value is made in line 8.

When SPF explores either of the paths, it uses a constraint solver (line 13 and 20) to determine whether the updated path condition is satisfiable or not, which instructs SPF to continue exploration of the path or backtrack to a previous choice, respectively. Note that a path condition choice is introduced regardless of path satisfiability of the updated path conditions of both branches. In practice, it happens frequently that some execution paths are infeasible caused by e.g. semantic dependencies that always hold prohibiting some branches in subsequent if-statements to be taken. When the path condition of only one branch is satisfiable, introducing a path condition choice is clearly superfluous and moreover adds to the state space JPF needs to explore. Consequently, analysis times are affected due to the larger state space and re-execution of branching instructions.

In SPF v7, we address this by incorporating an optimization that determines the feasibility of the branches in an *a priori* fashion instead of *posteriori*. When the path conditions of multiple branches are satisfiable (two for e.g. IFGT, and *at least* two for e.g. LCMP), a choice is introduced, otherwise, if only one is satisfiable, JPF is instructed to explore that particular path. Note also in this case that the path condition needs not be updated with the decision predicate associated with the choice, because it is subsumed, thus leading to a reduced set of constraints in the path condition compared to the original approach. In the event that none are satisfiable, JPF backtracks. A fragment of the algorithm is shown in Listing 2 for the IFGT Java Bytecode.

```
1 if(!ti.isFirstStepInsn()) {
2   ...
3   boolean gtSat = gtPC.solve();
4   boolean leSat = lePC.solve();
5   if(gtSat) {
6     if(leSat) {
7       cg = new PCChoiceGenerator(2);
8       sysState.setNextChoiceGenerator(cg);
9       return this;
10    } else {
11      return getTarget();
12    }
13  } else {
14    if(!leSat)
15      sysState.setIgnored(true);
16    return getNext(ti);
17  }
18 } else {
19   ...
20   boolean conditionValue = (Integer)cg.
         getNextChoice()==1 ? true : false;
21   if(conditionValue) {
22     pc._addDet(Comparator.GT, sym_v, 0);
23     cg.setCurrentPC(pc);
24     return getTarget();
25   } else {
26     pc._addDet(Comparator.LE, sym_v, 0);
27     cg.setCurrentPC(pc);
28     return getNext(ti);
29   }
30 }
```

Listing 2: Optimized approach for introducing path condition choices.

Again, the condition in line 1 will evaluate to true upon the first execution of the instruction. This time, however, the path conditions of the branches are solved a priori; line 3 and 4 determine their satisfiability and a path condition will be created in line 7 if both are satisfiable. Otherwise, execution continues.

2.2 Results

We demonstrate the effect of the optimization using the Bank Account example (see Listing 3 for an excerpt); a series of deposits and withdrawals are made to a bank account, and the action to perform is based on a symbolic variable `deposit`. The example features infeasible paths (line 5 and 9), deliberately inserted to demonstrate the effect of the optimization.

```
1 for(int i = 0; i < seqs; i++) {
2   boolean deposit = performDeposit();
3   if(deposit) {
4     b.deposit(10);
5     if(!deposit)
6       b.withdraw(1);
7   } else {
8     b.withdraw(1);
9     if(deposit)
10      b.deposit(10);
11  }
12 }
```

Listing 3: Excerpt of the Bank Account example.

The Bank Account example has been evaluated on a machine featuring an Intel Core i7 @ 2.70 GHz and 8 GB of memory.

The results in terms of analysis time and number of states explored when gradually increasing the sequence length are shown in Table 1. The significant reduction in the state space size is

Seq., [#]	Analysis time, [s]		Δ	States, [#]		Δ
	No Opt.	Opt.		No Opt.	Opt.	
15	17	13	24%	196,603	65,535	67%
16	30	22	27%	393,211	131,071	67%
17	61	41	33%	786,427	262,143	67%
18	126	84	33%	1,572,859	524,287	67%

Table 1: Performance comparison between SPF v6 and SPF v7 on the Bank Account example.

attributed the fact that 67% of the choices are infeasible thus yielding that 67% fewer states are introduced using the new optimization. Clearly, this example is dominated by infeasible paths, but it demonstrates that analysis times are reduced using the optimization. We also evaluated the effect of the optimization on the Red Black Tree example included as part of the SPF distribution. Here the number of states were reduced from $16,549$ to $4,067$, a 75% decrease in states. However, the analysis time reduction is relatively negligible going from $90.3s$ to $88.0s$ on average, a result that can be attributed that the analysis is dominated by the constraint solver.

3. SYMBOLIC EXECUTION TREE EXTENSION

SPF v7 is extended with the capability of producing a symbolic execution tree data structure. A symbolic execution tree characterizes the feasible execution paths of the program during symbolic execution. Nodes represent program states, and edges represent the transition from a state to another. Providing the actual symbolic execution tree after symbolic execution gives an intermediate structure that can be used for various post-processing facilities and/or structure-based analyses and transformations. This section describes the new extension, and elaborates on how it can be leveraged in other analyses. In Section 4, we describe a concrete application of the facilities the extension provides.

A high-level class-diagram of the extension is shown in Figure 1. The entry point of the extension is the `ASymbolicExecutionTreeListener`; an implementation of `PropertyListener` from JPF-core. The listener uses the `SymbolicExecutionTreeGenerator` for constructing the tree, and the actual construction process is chained to the events occurring during symbolic execution. The simplest case is the event occurring whenever an instruction, part of the symbolic target method or in its call chain, is executed. In that case, the `NodeFactory` will be requested to construct a new node which is connected to the node of the previously executed instruction. However, other events need to be taken account; an example is state backtracking signalling that a new sub-tree is generated at the choice to which JPF is backtracking.

The use of the `NodeFactory` (realizing the factory design pattern) allows for customizing the granularity level of the data included in the tree; for applications processing the complete symbolic traces, nodes corresponding to all executed instructions along the paths could be included, whereas other applications processing only recorded path conditions updates, could include only the nodes for branching instructions. The size of the resulting symbolic execution tree is thus largely dependent on the granularity level since it grows exponentially in the number of branches. A default node factory is provided, which produces nodes for each symbolic state.

3.1 Using the Extension

Application of the new extension requires two parts; a subclass of `ASymbolicExecutionTreeListener` implementing the two abstract methods; `getNodeFactory` and `processSET`. The former must provide an instance of `NodeFactory`, which will be used during the construction process to incorporate the information of interest in the nodes of the tree. The latter method will automatically be called upon completion of constructing the tree. In case multiple (symbolic) target methods are supplied in the SPF configuration, a respective tree is generated for each of them. Thus `processSET` provides the entry point for extensions that wish to conduct post-processing of the symbolic execution tree(s).

Processing the symbolic execution tree can be done in a variety of ways, but the extension makes available a simple framework for traversal-based analysis using the visitor pattern; user-defined extensions wishing to use this, need to implement `SymbolicExecutionTreeVisitor` and make definitions for the `visit` methods for each tree element.

4. APPLICATIONS OF THE SYMBOLIC EXECUTION TREE EXTENSION

The new extension is an appropriate format for analyses reasoning on all the execution paths produced as part of symbolic execution or when an intermediate representation is more appropriate for leveraging analysis.

In this section, we present SPF-Visualizer; a new addition to the SPF toolset, which visualizes the symbolic execution tree thus being useful for e.g. debugging or teaching purposes. It supports a wide variety of the most common output formats, including DOT, PS, PNG, PDF, and more. SPF-Visualizer is generically built, and as such, allows for easy tailoring of the visual output both in terms of *what* should be represented and *how*. Here we demonstrate a basic application of SPF-Visualizer and give pointers for how to adopt it for more domain-specific purposes. Specifically, we want to output the symbolic execution tree in terms of the complete symbolic execution paths graphically; the nodes in the graphic representation include basic information about the symbolic state, such as the Program Counter (i.e. the instruction to be executed when firing and outgoing transition), and references to the original source code including line number and class to which the Program Counter belongs. Note that this is a fine-grained visualization, that may not scale to large systems, but serves to demonstrate the possibilities of SPF-Visualizer-other applications may e.g. only be interested in visualizing path condition updates, thus being far more scalable.

Besides, we distinguish between the type of node and provide type-specific information:

Conditional goto instructions This includes Java Bytecodes such as IFEQ and IFGE. These are represented as a diamond shape inspired by the shape of decisions in flowchart diagrams.

Invoke instructions This includes INVOKEVIRTUAL and INVOKESTATIC. For making explicit that subsequent nodes will be part of the method body of the callee, these nodes are highlighted and furthermore the node includes information about the callee e.g. its name.

Return instructions This includes e.g. IRETURN and ARETURN. Similar to the invoke instructions, these are highlighted to make clear that flow of control returns to the callsite.

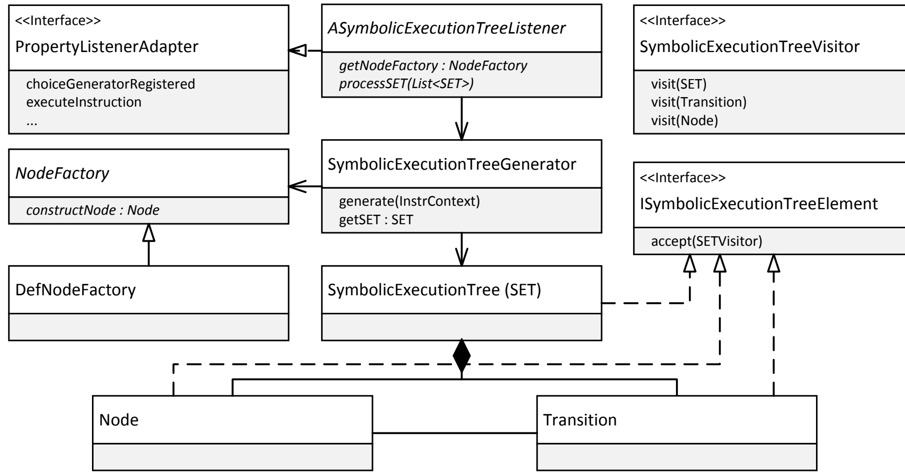

Figure 1: High-level class diagram of the Symbolic Execution Tree Extension.

Path condition updates This applies for the class of nodes for which the previous node is a conditional goto instruction where the conditional expression is symbolic. For such nodes, the entire conjunctive expression of the path condition is included in the node, and thus expresses the constraint that must be satisfied to follow that (part of the) execution path.

Final instructions This applies to the last instruction in each of the feasible execution paths symbolic execution has covered. In addition to information included due to the instruction being part of the above, these nodes also contains the final path condition.

Applying such type-specific renderings is easy, because SPF-Visualizer encloses their processing in distinct methods e.g. `getNodeAttrs(IfNode node)` for the nodes representing conditionals in the tree. Each method returns a set of attributes, e.g. shape, color, and text, that must apply for the particular node. It is thus relatively easy to make domain-specific visualizations. E.g. if one is concerned with the state of the operand stack during symbolic execution, such information can easily be incorporated as well because the `Node` object already contains such information. If the needs are beyond the knowledge that can be inferred from the symbolic state, one can supply an appropriate implementation of `NodeFactory` to SPF-Visualizer (recall, it is a subclass of `ASymbolicExecutionTreeListener`), which includes this information when constructing nodes.

We demonstrate the use of the basic SPF-Visualizer using the small program shown in Listing 4.

```
13 public int compAB(int a, int b) {
14   if(a > b) {
15     if(a == b) {
16       return number() + 42;
17     } else
18       return 42;
19   } else
20     return number();
21 }
22 public int number() {
23   return 24;
24 }
```

Listing 4: Java program for demonstrating the SPF-Visualizer.

The target method is `compAB` where both parameters are symbolic. Note that the conjunction of the branching conditions of line 14 and 15 is not satisfiable. Consequently, the execution path to line 16 is infeasible. The resulting visualization of the symbolic execution tree is shown in Figure 2.

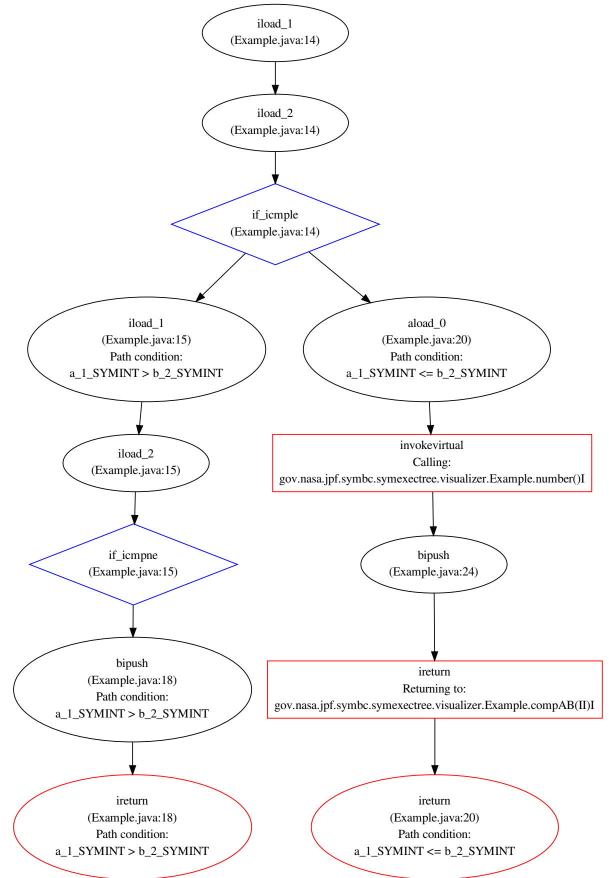

Figure 2: Visualization of the symbolic execution tree of the code in Listing 4.

From the visualization, it is clear that the execution path to line 16 is infeasible since the node of the branching instruction in line 15 only has a single outgoing edge leading to the node of line 18.

5. CONCLUSION

This paper has outlined SPF v7 in terms of its changes to address JPF-core v7. SPF v7 includes a new optimization for reducing the number of path condition choices, and the evaluation has demonstrated that it affects both state space size and analysis times.

We have also presented the Symbolic Execution Tree Extension; an extension to the SPF toolset that allows for producing a symbolic execution tree characterizing the execution paths covered as part of symbolic execution. SPF-VISUALIZER is an application of Symbolic Execution Tree Extension which outputs the symbolic execution tree graphically. It is generically designed for addressing various domain-specific visualizations by allowing adjustments in both *what* shall be represented as well as *how*. The new extension is also forming the core component in our current work [9] of translating the symbolic execution tree to the Timed Automata formalism of UPPAAL [2] with integration in TetaSARTS [6] for real-time analyses. A final direction is to adopt the extension for memoization purposes similar to [10].

In the context of what has been shown in this paper, future work comprises how to further optimize the code, e.g. in terms of the lazy initialization algorithm used in SPF for handling input data structures. Similar to the optimization presented for handling branch conditions, we would like to explore ways of delaying and reducing the non-determinism used for handling aliasing in the input data structures. Furthermore, we would like to improve the visual output of the tool. More sophisticated features such as dynamically allowing "collapsing" method bodies may provide a better overview of the tree.

6. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, 2007.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III: verification and control*, 1996.

[3] Choco. Choco, 2013. `http://www.emn.fr/z-info/choco-solver/`.

[4] JPF. Java PathFinder, 2013. `http://babelfish.arc.nasa.gov/trac/jpf`.

[5] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.

[6] K. S. Luckow, T. Bøgolm, B. Thomsen, and K. G. Larsen. TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems. In *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2013.

[7] J. M. Rojas and C. S. Păsăreanu. Compositional Symbolic Execution through Program Specialization. In *BYTECODE'13*, 2013.

[8] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving Complex Constraints for Symbolic PathFinder. In *Proceedings of the 3rd int'l conference on NASA Formal methods*, 2011.

[9] SPF-RT. Symbolic PathFinder Real-Time (jpf-symbc-rt), 2013. `http://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc-rt`.

[10] G. Yang, S. Khurshid, and C. S. Păsăreanu. Memoise: a Tool for Memoized Symbolic Execution. In *Proceedings of the International Conference on Software Engineering*, 2013.