Symbolic PathFinder: Symbolic Execution of Java Bytecode

Corina S. Păsăreanu and Neha Rungta
NASA Ames Research Center, Moffett Field, CA 94035, USA
{corina.s.pasareanu,neha.s.rungta}@nasa.gov

ABSTRACT

Symbolic Pathfinder (SPF) combines symbolic execution with model checking and constraint solving for automated test case generation and error detection in Java programs with unspecified inputs. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as constraints generated from the analysis of Java bytecode. The constraints are solved using off-the shelf solvers to generate test inputs guaranteed to achieve complex coverage criteria. SPF has been used successfully at NASA, in academia, and in industry.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*

General Terms

Reliability,Verification

Keywords

Automated test case generation, program analysis

1. INTRODUCTION

Symbolic execution is a popular analysis technique that executes a program on symbolic, rather than concrete, inputs and it computes the program effects by manipulating expressions in terms of these symbolic inputs. Symbolic execution [6] was introduced in the 70s, but only recently has found wider applicability in practice due to the availability of new powerful decision procedures (necessary for manipulating symbolic expressions) and increased computation power.

We present Symbolic Pathfinder (SPF)–a tool for performing symbolic execution of Java bytecode. SPF handles inputs and operations on booleans, integers, reals, and complex data structures with a polymorphic class hierarchy. It handles preconditions as well as multi-threading. Furthermore, SPF supports a mixed mode execution [7] that combines concrete and symbolic execution. SPF also offers preliminary support for String and bit-vector operations.

SPF has been used at NASA [7], to uncover subtle bugs in flight software, in academia, to aid in various research projects, and in industry, more recently at Fujitsu, to test web applications with over 60,000 SLOC. By solving the symbolic input constraints for various coverage obligations, SPF can been used as a customizable test generator. The user can specify different code coverage metrics (e.g. MC/DC), she can customize the search strategy for generating test cases, and she can save the tests in different formats, such as HTML tables or JUnit tests. Furthermore, SPF has been used to generate counterexamples to safety properties in concurrent programs with unspecified inputs [8] and for proving light-weight properties of software. SPF is a freely available open-source project [4].

Related Tools Unlike our previous work [1, 5], SPF does not require a program instrumentation and a type-based analysis, and hence it is more efficient. Bogor/Kiasan [2], unlike SPF, does not separate between concrete and symbolic data, hence it can not support mixed concrete/symbolic execution. Furthermore, it can not handle complex Math constraints. Also related are concolic tools [9, 3], which perform a form of symbolic execution along concrete program paths. The tools work by program instrumentation and do not handle multi-threading systematically.

2. TOOL DESCRIPTION

SPF is part of the Java PathFinder verification tool-set [4]. Java Pathfinder includes JPF-core, an explicit-state model checker, and several extension projects, one of them being SPF (`jpf-symbc` Java project). The model checker consists of a an extensible custom Java Virtual Machine (VM), state storage and backtracking capabilities, different search strategies, as well as *listeners* for monitoring and influencing the search. JPF-core executes the program concretely based on the standard semantics of the Java.

In contrast, SPF replaces the concrete execution semantics of JPF-core with a non-standard symbolic interpretation.

SPF relies on the JPF-core framework to systematically explore the different 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. We describe SPF's features below (see Fig. 1).

Instruction Factory and Attributes SPF replaces the standard concrete execution semantics by using a `SymbolicInstructionFactory`, that extends the bytecode instructions to manipulate symbolic values and expressions. For example, when adding two symbolic integers sym_1 and sym_2 (by executing the `IADD` bytecode) the result is a symbolic expression representing $sym_1 + sym_2$.

Storage of symbolic values and expressions is accomplished

ASE'10, September 20–24, 2010, Antwerp, Belgium.
ACM 978-1-4503-0116-9/10/09.

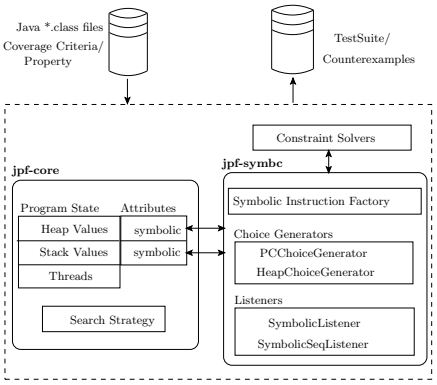

Figure 1: Architecture and features of SPF

by assigning symbolic attributes to variables, fields and stack operands. The attributes are not part of the (concrete) program state and thus it is possible to use both concrete and symbolic values during the same execution [7]. This can be used, for example, to first perform a concrete execution of the program to reach a "suspicious" state, from which point on one can perform a detailed symbolic execution to stress that state [7]. Furthermore, it allows for easy extension with other analyses that maintain both concrete and symbolic data such as concolic execution [3].

Branching Conditions The symbolic execution of conditional instructions (`if` statements) involves exploration of paths corresponding to the predicate at the branch evaluating to *true* and *false*. Both choices are generated non-deterministically by the `PCChoiceGenerator`. Each generated choice is associated with a path condition encoding the condition or its negation respectively. The path conditions are checked for satisfiability using off-the-shelf decision procedures or constraint solvers. If the path condition is satisfiable, the search continues; otherwise, the search backtracks (meaning that branch is unreachable).

Decision Procedures/Constraint Solvers SPF uses multiple decision procedures and constraint solvers through a generic interface. Currently, SPF supports: `choco` for integer/real constraints, `cvc3` for linear constraints, and the interval arithmetic solver `IASolver`. Adding support for additional constraint solvers such as `HAMPI` and `YICES` is work in progress.

Handling Input Data Structures SPF uses *lazy initialization* [5] to handle unbounded input data structures. The execution starts on data structures with un-initialized fields and it initializes them lazily, when the fields are first accesed. A field of class T is initialized non-deterministically to (1) null, (2) a reference to a new instance of class T with uninitialized fields, or (3) a reference to an object of type T created during a prior field initialization; this sytematically treats aliasing. The `HeapChoiceGenerator` is used to generate the choices. We have recently extended SPF to provide support for polymorphism. Step (2) above is replaced with non-deterministically assigning new instances of class T and of all the classes that *inherit* from T. Similarly, step (3) is replaced with assigning previously created objects to class T and objects from classes that *inherit* from T.

Handling Math Functions SPF uses JPF-core's native peers mechanism to model native libraries and any other program parts that cannot be analyzed directly with symbolic execution. Most notably, SPF incorporates native peers that capture the calls to the `java.lang.Math` libraries and dispatch them to an appropriate constraint solver that can handle complex Math constraints. The same mechanism is also used for capturing String operations.

Symbolic Listeners The listeners gather and display information about the path conditions generated during the symbolic execution. They generate test cases and sequences in various formats.

	Error Type	SLOC	States	Time	Memory
VecDeadlock0	Deadlock	7267	1370	4.56s	66 MB
VecDeadlock1	Deadlock	7169	2948	6.89s	69 MB
VecRace	Race	7151	3120	7.98s	65 MB

Table 1: SPF Results

3. RESULTS AND CONCLUSIONS

Table 1 gives the resources consumed for using SPF to detect two deadlocks and race-condition in the `Vector` class in the JDK 1.4 library [8].

We presented Symbolic Pathfinder, a symbolic execution tool for automatic test case generation and error detection in Java programs. Although effort was put in optimizing the code, the tool suffers from scalability issues due to the exhaustive nature of the analysis it performs and the constraint solving involved. Towards this end, we are working on parallelizing SPF [10].

Acknowledgments We would like to thank the people who contributed to the tool: Hank Bushnell, Peter Mehlitz, Suzette Person, Matt Staats, and Willem Visser.

4. REFERENCES

[1] S. Anand, C. Păsăreanu, and W. Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. *TACAS*, pages 134–138, 2007.

[2] X. Deng, J. Lee, and Robby. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, 2006.

[3] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[4] Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf.

[5] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Proc. TACAS*, pages 553–568, 2003.

[6] J. C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[7] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. ISSTA*, 2008.

[8] N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. *SPIN*, 2009.

[9] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.

[10] M. Staats and C. Pasareanu. Parallel Symbolic Execution for Structural Test Generation. In *ISSTA*. ACM, 2010.