JPF–SE: A Symbolic Execution Extension to Java PathFinder

Saswat Anand[1], Corina S. Păsăreanu[2], and Willem Visser[2]

[1] College of Computing, Georgia Institute of Technology
saswat@cc.gatech.edu
[2] QSS and RIACS, NASA Ames Research Center, Moffett Field, CA 94035
{pcorina,wvisser}@email.arc.nasa.gov

Abstract. We present JPF–SE, an extension to the Java PathFinder Model Checking framework (JPF) that enables the symbolic execution of Java programs. JPF–SE uses JPF to generate and explore symbolic execution paths and it uses off-the-shelf decision procedures to manipulate numeric constraints.

1 Introduction

Explicit state model checking tools, such as Java PathFinder (JPF) [5, 12], are becoming effective in detecting subtle errors in complex concurrent software, but they typically can only deal with closed systems. We present here JPF–SE, a symbolic execution extension to Java PathFinder, that allows model checking of concurrent Java programs that take inputs from unbounded domains.

JPF–SE enables symbolic execution of Java programs during explicit state model checking, which has the following unique characteristics: (a) checks the behavior of code using symbolic values that represent data for potentially infinite input domains, instead of enumerating and checking for small concrete data domains (b) takes advantage of the built-in capabilities of JPF to perform efficient search through the program state space: systematic analysis of different thread interleavings, heuristic search, state abstraction, symmetry and partial order reductions (c) enables modular analysis: checking programs on un-specified inputs enables the analysis of a compilation unit in isolation (d) automates test input generation for Java library classes [13] (e) uses annotations in the form of method specifications and loop invariants to prove light-weight properties of Java programs [8] and (f) uses a common interface to several well-known decision procedures to manipulate symbolic numeric constraints; JPF–SE can be extended easily to handle other decision procedures.

2 JPF–SE Overview

Java PathFinder. JPF [5, 12] is an explicit-state model checker for Java programs that is built on top of a customized Java Virtual Machine. By default,

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 134–138, 2007.

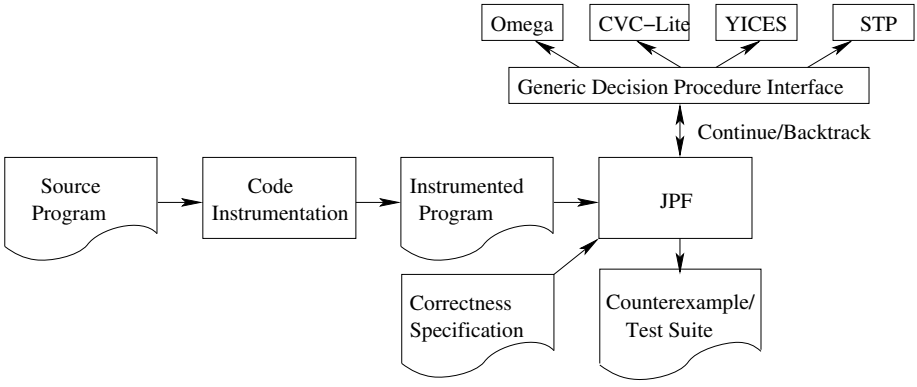

Fig. 1. Tool Architecture

JPF stores all the explored states, and it backtracks when it visits a previously explored state. The user can also customize the search (using heuristics) and it can specify what part of the state to be stored and used for matching.

Symbolic Execution. Symbolic execution [7] is a technique that enables analysis of programs that take un-initialized inputs. The main idea is to use *symbolic values*, instead of actual (concrete) data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* and a program counter. The path condition accumulates constraints which the inputs must satisfy in order for an execution to follow the corresponding path.

JPF–SE Architecture. In previous work, we presented a framework that uses JPF to perform symbolic execution for Java programs [6, 8]. It has now been added to the JPF open-source repository [5] and is illustrated in Figure 1. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure. If the path condition is unsatisfiable, the model checker backtracks. The approach can be used for finding counterexamples to safety properties and for test input generation (that satisfy a testing criterion, such as branch coverage).

Symbolic State Space Exploration. JPF–SE exploits JPF's ability to explore arbitrary program control flow (loops, recursion, method invocation), but performing symbolic execution on a program with loops (or recursion) may result in an infinite number of symbolic states. JPF–SE uses two complementary techniques to address this problem: (a) for systematic state space exploration JPF–SE puts a *bound* on the size of the program inputs and/or the search depth,

Table 1. Comparative Results. "N/A" indicates not supported.

Example	Interface	Omega	CVCL	YICES	STP
TCAS	File	00:15	00:26	N/A	N/A
	Pipe	00:04	00:12	N/A	N/A
	Native	00:03	00:13	00:06	00:31
	Native table	00:01	00:11	00:05	N/A
	Native inc	N/A	00:03	00:01	N/A
TreeMap	File	02:02	06:02	N/A	N/A
	Pipe	07:42	13:04	N/A	N/A
	Native	01:39	06:11	03:06	>60:00
	Native table	00:40	05:10	02:36	N/A
	Native inc	N/A	02:58	00:33	N/A

and (b) JPF–SE provides automated tool support for *abstracting* and *comparing* symbolic states, to determine if a symbolic state has been visited before, in which case the model checker will backtrack (see [1] for details).

Decision Procedures. JPF–SE uses the following decision procedures; they vary in the types of constraints they can handle and their efficiency. **Omega library** [9] – supports linear integer constraints. **CVC-Lite** [3] – supports integer, rational, bit vectors, and linear constraints. **YICES**[1] [4] – supports types and operations similar to those of CVC-Lite. **STP**[2] [2] – supports operations over bit vectors. In the JPF–SE interface, all integers are treated as bit vectors of size 32. Recently, we have also added a constraint solver, **RealPaver** [10], that supports linear and non-linear constraints over floating point numbers.

Generic Decision Procedure Interfaces. JPF–SE provides three interfaces with decision procedures. They vary in their degree of simplicity and efficiency. In the **file** based interface, the decision procedure is started for each query and a query is sent (and result received) via a file. This interface is the simplest to use and extend, but in general it is slow. With the **pipe** interface, the decision procedure is run concurrently with JPF and the communication is accomplished over a pipe. Although this does not suffer the process startup cost of the file approach it is harder to use and extend and it is operating system and language specific. With the **native** interface, JPF communicates directly with the decision procedure through a Java Native Interface (JNI). This mode is most difficult to implement among the three, but is usually much faster.

There are two optimizations available for the native interface: a table-based approach for efficient storing of the path condition that allows sharing of common sub-expressions and if the decision procedure supports incremental constraint analysis, the path condition is not sent all at once but rather just the new constraint that should be added/removed before checking satisfiability.

[1] SMT competition 2006 winner in all categories but one.
[2] SMT competition 2006 winner for QF_UFBV32 (Quantifier Free, Uninterpreted Functions, Bit Vector).

Experience with Different Decision Procedures. The interfaces for communications with the decision procedures is defined such that it is straightforward to connect a new tool. As a consequence, JPF–SE is well suited for performance comparisons across a wide array of examples. We show in Table 1 the runtime results (in mins:secs) for generating all reachable states while running JPF–SE with varying decision procedure configurations over two examples: TCAS from the Siemens Suite and on the TreeMap example from [13]. TCAS is small (only 2694 queries) but contains many constraints that are both satisfiable and unsatisfiable; TreeMap produces many queries (83592), but they are all satisfiable.

The preliminary results indicate that the native interfaces are the fastest and both the optimizations (where applicable) improve the performance further. For this reason YICES and STP are only used through the native interface.

3 Conclusion and Future Work

We have presented JPF–SE, an extension to JPF that enables symbolic execution of Java programs to be performed during model checking. JPF–SE uses JPF to generate and explore symbolic states and it uses different decision procedures to manipulate numeric constraints. JPF–SE has been applied to checking concurrent Java programs and to generating test inputs for Java classes. In the future we plan to extend JPF–SE's code instrumentation package, which currently handles only numeric values, to handle symbolic complex data structures. We also plan to add compositional reasoning for increased scalability and to interface with tools using the SMT-LIB standard [11] (through file and pipe).

Acknowledgements

We thank Sarfraz Khurshid and Radek Pelánek for contributing to this work.

References

1. S. Anand, C. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. SPIN*, 2006.
2. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Computer and Comm. Security*, 2006.
3. CVCL. http://www.cs.nyu.edu/acsys/cvcl/.
4. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of CAV*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.
5. Java PathFinder. http://javapathfinder.sourceforge.net.
6. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS'03*, Warsaw, Poland, April 2003.
7. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.
8. C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc of SPIN'04*, volume 2989 of *LNCS*, 2004.

9. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.
10. realPaver. http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/.
11. SMT-LIB. http://combination.cs.uiowa.edu/smtlib/.
12. W. Visser, K. Havelund, G. Brat, S. J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
13. W. Visser, C. Pasareanu, and R. Pelanek. Test input generation for java containers using state matching. In *Proc. ISSTA*, 2006.