Symbolic Pathfinder for SV-COMP
(Competition Contribution)

Yannic Noller[1], Corina S. Păsăreanu[2,3], Aymeric Fromherz[2], Xuan-Bach D. Le[2], and Willem Visser[4](✉)

[1] Humboldt-Universität zu Berlin, Berlin, Germany
[2] Carnegie Mellon University Silicon Valley, Moffett Field, USA
[3] NASA Ames Research Center, Mountain View, USA
[4] Stellenbosch University, Stellenbosch, South Africa
visserw@sun.ac.za

Abstract. This paper describes the benchmark entry for Symbolic Pathfinder, a symbolic execution tool for Java bytecode. We give a brief description of the tool and we describe the particular run configuration that was used in the SV-COMP competition. Furthermore, we comment on the competition results and we outline some directions for future work.

1 Verification Approach

Symbolic Pathfinder (SPF) is a program analysis tool for Java bytecode; the tool is based on *symbolic execution*. In this approach, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from the analysis of the code structure, i.e. conditionals and other statements in the program. These constraints are then solved using different constraint solvers (both off-the-shelf and built-in-house) to generate test inputs that are guaranteed to reach those parts of the code.

The current implementation handles the following:

- Inputs of type boolean, int, long, float, double
- Input data structures, using *lazy initialization* [5]
- Preconditions [5]
- Multi-threading (via Java PathFinder exploration)
- Mixed symbolic/concrete execution mode [9]
- Symbolic arrays [3]
- Inputs of type String – work in progress [1].

SPF can also be used for probabilistic analysis by leveraging model counting over symbolic constraints [2,4], and for automated program repair [6,7]. Most recent work explores combinations of SPF with AFL-style fuzzing [8] and further differential analysis for regression problems.

https://doi.org/10.1007/978-3-030-17502-3_21

2 Software Architecture

SPF is described in detail in a journal article [10] (however, as it is an active project, it is being updated with new features all the time). We depict the current tool architecture in Fig. 1. The input to the tool is a Java bytecode program and a configuration file that specifies different options for analysis (as discussed below). The output is a set of test sequences that execute different paths through the code. The output also lists the errors that were found (e.g. exceptions, assert violations) together with various statistics about the analysis.

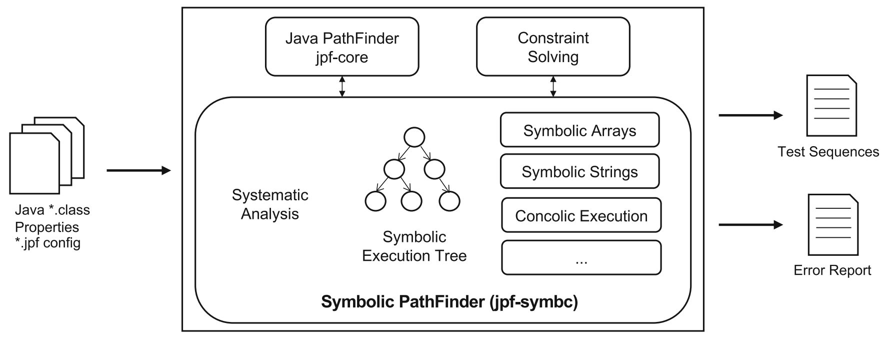

Fig. 1. Symbolic PathFinder overview.

Symbolic execution is implemented by a "non-standard" interpretation of bytecodes. The symbolic information is propagated via *attributes* associated with program variables, operands, etc. The analysis can start from any point in the program and it can perform mixed concrete/symbolic execution. SPF relies on jpf-core's search engine to explore different paths through the code. The default search strategy is depth-first search. State matching (as implemented in jpf-core) is usually turned off during symbolic execution.

SPF uses several constraint solvers and decision procedures, most notably Z3 and Z3bitvector, which are available from https://github.com/Z3Prover/z3. SPF implements both incremental and non-incremental constraint solving.

3 Discussion of Strengths and Weaknesses of the Approach

The competition results are provided on the SV-COMP website. The results indicate that SPF outperforms the other tools in terms of correct answers (337), cpu time (1300 s) and energy (13000 J). However, SPF also reported 6 incorrect results, which penalized the overall final score. While the incorrect true results are due to the bounded nature of the analysis, the incorrect false results are due mainly to the string analysis, with the exception of one result which was due to an error in jpf-core which has since been corrected. The string solver was incorrectly specified and tested (i.e. the path to the string solver is hard coded in the current implementation but we provided no string solver for the competition).

In the future we plan to test SPF on the competition string examples using either ABC or Z3str and to robustify the implementation. We also plan to contribute to the competition by adding more interesting benchmarks, particularly related to input data structures.

4 Tool Setup and Configuration

Symbolic PathFinder is available at https://github.com/SymbolicPathFinder/jpf-symbc. It requires Java 8 and Java PathFinder, which available at https://github.com/javapathfinder/jpf-core.

For this competition we used the version with the timestamp *Mon Nov 19 09:51:16 CET 2018*, which refers to the date when we pulled the artifacts from the GitHub repository and generated the jpf-symbc jar archive.

To run SPF, the user needs to download Symbolic PathFinder and Java PathFinder (default branches) and create a file `.jpf/site.properties` in the home directory. The `site.properties` file should contain the following lines (the users should modify to point to the location of `jpf-core` and `jpf-symbc` on their computer):

```
jpf-core = ${user.home}/workspace/jpf-core
jpf-symbc = ${user.home}/workspace/jpf-symbc
extensions = ${jpf-core},${jpf-symbc}
```

The user then creates a `*.jpf` configuration file (described in detail below). For the competition we modified the SPF tool to handle the non-deterministic constructs required by the competition.

4.1 Example Configuration

We give here an example configuration that can be used to run the SPF tool; this is the default configuration, that we used in the competition. The explanation for the different options is given in parenthesis.

- `target=test.Main` *(specify the target application)*
- `classpath=/..` *(path to your class example)*
- `sourcepath=/..` *(path to the source of your example)*

- `symbolic.dp=z3bitvector` *(specify the decision procedure)*
- `symbolic.bvlength=64` *(specify the bitvector length)*

- `symbolic.min_int=-100` *(specify various min max values)*
- `symbolic.max_int=100`

- `symbolic.min_double=-100.0`
- `symbolic.max_double=100.0`

- `symbolic.debug=true` *(print debug information)*
- `search.depth_limit=15` *(specify search limit)*

- `symbolic.lazy=on` *(handling symbolic arrays)*
- `symbolic.arrays=true`

- `symbolic.strings=true` *(specify string analysis)*
- `symbolic.string_dp=ABC` *(specify string solver).*

SPF also has the option of running the constraint solving incrementally. Note however that we did not use the string solving and the incremental solving options in the competition as we did not have enough time to prepare and test those features, as we were entered late in the competition.

5 Software Project and Contributors

Information about the project and contributors can be found at the project webpage: https://github.com/SymbolicPathFinder/jpf-symbc. For more information please contact the authors of this paper.

References

1. Bang, L., Aydin, A., Phan, Q., Pasareanu, C.S., Bultan, T.: String analysis for side channels with segmented oracles. In: FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 193–204 (2016)
2. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in symbolic PathFinder. In: ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 622–631 (2013)
3. Fromherz, A., Luckow, K.S., Pasareanu, C.S.: Symbolic arrays in symbolic PathFinder. ACM SIGSOFT Softw. Eng. Notes **41**(6), 1–5 (2016)
4. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: ISSTA 2012, Minneapolis, MN, USA, 15–20 July 2012, pp. 166–176 (2012)
5. Khurshid, S., PĂsĂreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
6. Le, X.-B.D., Chu, D.-H., Lo, D., Le Goues, C., Visser, W.: JFIX: semantics-based repair of Java programs via symbolic PathFinder. In: ISSTA 2017, pp. 376–379 (2017)
7. Le, X.-B.D., Chu, D.-H., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: ESEC/FSE 2017, pp. 593–604. ACM, New York (2017)
8. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: complexity analysis with fuzzing and symbolic execution. In: ISSTA 2018, Amsterdam, The Netherlands, 16–21 July 2018, pp. 322–332 (2018)

9. Pasareanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: ISSTA 2011, Toronto, ON, Canada, 17–21 July 2011, pp. 34–44 (2011)
10. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013)