Using Symbolic (Java) PathFinder at NASA

Corina S. Păsăreanu

Carnegie Mellon University/NASA Ames Research Center, Moffett Field, CA 94035, USA,
corina.s.pasareanu@nasa.gov

Abstract. Symbolic (Java) PathFinder (SPF) is a symbolic execution tool that is used for the automated generation of test cases that satisfy various coverage criteria, such as condition, path and MC/DC coverage. The tool has been used at NASA, in academia, and in industry, most notably at Fujitsu. We describe our experience with using SPF and we identify some of the challenges for making the tool more usable. While our discussion focuses on SPF, we believe that the challenges described here apply to other advanced verification tools as well.

1 Introduction

Modern software involves programming multi-core processors, cloud-based systems, and cyber-physical systems. Such software needs to be highly reliable and secure. Techniques for ensuring reliability and security of software include testing and formal analysis, such as theorem proving, model checking, symbolic execution, static analysis, etc. Despite the advances in formal techniques, testing is still the most widely used method for error detection at NASA, as well as elsewhere. However, while formal techniques can provide exhaustive analysis with strong guarantees of reliability and safety, testing may miss important errors, since it can only analyze *some* of the program executions. In our work, we aim to use the strengths of automated formal techniques to make testing more effective.

2 Symbolic PathFinder

Towards this end, we have developed Symbolic PathFinder (SPF), a tool that integrates symbolic execution, with model checking and constraint solving to perform automated generation of test cases, while checking properties of code during test case generation. SPF uses the analysis engine of Java PathFinder (JPF) [1] - a software model checking tool developed at NASA Ames. We have found that generating test cases is potentially the easiest path towards the infusion of advanced technologies, such as model checking and symbolic execution.

In SPF, programs are executed on symbolic inputs that represent all possible concrete inputs. Values of variables are represented as numeric (mixed integer and real) constraints, encoding the conditions from the code. These constraints are then solved to generate test inputs guaranteed to exercise the analyzed code.

SPF relies on off-the-shelf constraint solvers to solve the symbolic constraints. SPF handles pre-conditions, recursive input data structures as inputs and concurrency. The user can customize SPF: (i) to specify the testing coverage criteria (e.g. path, condition, MC/DC, user-defined), (ii) to pick different search strategies (breadth-first search, depth-first search, heuristic search), and (iii) to output the test cases in different formats, such as HTML, JUnit tests and a format accepted by NASAs Antares simulator.

Symbolic PathFinder is built on top of JPF and naturally analyzes *Java bytecode*. However, through a project at Ames, it now applies to different kinds of *models* as well [5]: e.g. Simulink/Stateflow, Standard UML, Rhapsody UML, etc. thus increasing SPF's usability. We have built translators from these models into a common Java environment, to make them amenable for test case generation with SPF. In our "polyglot" framework one can translate a state machine in Java and then plug-in different semantics, e.g. use a Stateflow interpreter or a UML interpreter, to study inter-operability, i.e. if a model created with one tool, e.g. Mathworks'Stateflow, can run and satisfy the same properties with a different tool, e.g. Rhapsody.

3 Experience

SPF has been applied successfully at NASA, in academia and in industry, most notably at Fujitsu, where it was used for testing web applications.

In one application [4], SPF was used to generate test cases for a flight software component, the On-board Abort Executive developed at NASA JSC. The executives role is to monitor a set of flight rules aboard a spacecraft. When the flight rules are violated, a mission abort is issued. For this application, SPF was customized to generate test cases in the form of Antares scripts, so that to be used directly by the developers in their testing environment. Furthermore, the search strategy to be used was customized to satisfy some specific requests from the developer, i.e. to consider only one flight rule violation at the time. While manual testing took one week and could not achieve adequate coverage, and guided random testing achieved poor coverage, SPF generated test cases that achieved complete coverage (all possible flight rule and abort combinations) in less than a minute. During test case generation, SPF also reported some property violations (e.g. a flight rule as broken but no abort was picked). Furthermore the generated test suite was used in regression testing for a new version of the executive, which helped discover a major bug that led to the re-design of the executive component and also of other components that interacted with it.

While performing this case study, we found some interesting issues related to the application of SPF for automated test case generation. For the on-board abort executive, the input data is constrained by the environment or physical laws. For example we generated test inputs, where inertial velocity is 24000 ft/s when the geodetic altitude is 0 ft. This is physically impossible. To avoid generation of such test cases, one would need to encode these constraints explicitly as pre-conditions, which would require non-trivial human input. Our solution was to use machine learning techniques on the system level simulation runs to infer such constraints automatically [2].

In conclusion, it is not enough to generate test cases that achieve full coverage; the test cases also need to make sense to the software developers. Furthermore, the assumptions about the environment of the components under analysis (i.e. the physical preconditions in our case) need to be made explicit (using inputs from the user and also automated inference techniques). This is true not only for SPF but for many other advanced tools that apply to software components (since the entire system under analysis is seldom available or amenable to formal verification).

4 Usability Challenges

Based on our experience with SPF and similar tools, we have identified the following (incomplete list of) challenges.

4.1 Tool Usability

- the tool should be easy to set-up (this is not true for JPF, and SPF) and portable for multiple architectures
- the tool should provide animation and/or graphical display of results, including intermediate results (e.g. coverage, nodes, transitions, etc); this is particularly important for tools that do not give "instant" responses, such as SPF
- the tool should provide informative counterexamples/explanations of errors;
- the tool documentation should be easily accessible on-line (the JPF project uses wiki pages)
- the tool should be integrated with other tools that are used by developers for writing and/or testing their code (e.g. JPF provides an Eclipse plug-in; SPF outputs test cases in the form of JUnit tests or Antares scripts)
- the tool should provide clear display of different options (this is not true for JPF/SPF that have many different options that are often confusing)
- the tool should be highly available – e.g. the JPF is open source

4.2 Research Challenges

- analysis should be as efficient as possible; symbolic execution techniques are still very slow/costly in terms of number of paths to explore and number of constraints to be solved; parallelization, compositional techniques and fast constraint solvers will provide solutions to this challenge
- powerful constraint solvers and decision procedures are needed; applications from NASA domain require solving complex, non-linear mathematical constraints that are undecidable or very hard to solve; new heuristic techniques are necessary to solve such problems; test case generation for web applications and security problems requires solving over string constraints and combinations of numeric and string constraints(a very active area of research)

- automatic discovery of interfaces is needed, perhaps with (minimal) inputs from the users; as mentioned the environment assumptions for the components under analysis need to be made explicit; automata and machine learning techniques [3] and their combinations, can be used to infer the calling context of the components

4.3 Domain Specific Knowledge

Finally we believe that for the successful adoption of advanced software analysis tools in industry, one would need to provide for easy integration of domain specific knowledge in the analysis and easy adaptation of the tool to the users specific needs. For example, NASA uses many different models (for model based development, for robot planning and execution, for integrated system health management, etc.) Therefore we adapted SPF to work from code to models. Also we customized the search strategies and output format of the tool based on the user's needs. Of course, there may be many other ways of incorporating the user's domain specific demands into an advanced analysis.

5 Conclusion

In this paper we discussed the Symbolic PathFiner tool, our experience with using it, and some lessons learned. We also outlined some of the challenges for adopting the tool in industry. Although we made our presentation in the context of a particular tool, we believe that our discussion is relevant for other advanced verification tools.

References

1. Jpf project, 2010. http://babelfish.arc.nasa.gov/trac/jpf.
2. M. Davies, C. S. Păsăreanu, and V. Raman. Symbolic execution enhanced system testing. In *Submitted for publication*, 2010.
3. D. Giannakopoulou and C. S. Păsăreanu. Interface generation and compositional verification in javapathfinder. In *Proceedings of FASE*, 2009.
4. 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 *ISSTA'08: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2008.
5. C. S. Păsăreanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karasai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *Proceedings of SMC-IT*, 2009.