Test Input Generation with Java PathFinder: Then and Now (Invited Talk Abstract)

Sarfraz Khurshid
University of Texas
Austin, USA
khurshid@ece.utexas.edu

Corina S. Păsăreanu
Carnegie Mellon University
Silicon Valley, NASA Ames
Research Center
Moffet Field, California
corina.pasareanu@west.cmu.edu

Willem Visser
Stellenbosch University
Stellenbosch, South Africa
wvisser@cs.sun.ac.za

ABSTRACT

The paper *Test Input Generation With Java PathFinder* was published in the International Symposium on Software Testing and Analysis (ISSTA) 2004 Proceedings, and has now been selected to receive the ISSTA 2018 Retrospective Impact Paper Award. The paper described black-box and white-box techniques for the automated testing of software systems. These techniques were based on model checking and symbolic execution and incorporated in the Java PathFinder analysis tool. The main contribution of the paper was to describe how to perform efficient test input generation for code manipulating complex data that takes into account complex method preconditions and evaluate the techniques for generating high coverage tests.

We review the original paper and we discuss the research that preceded it and the research that has happened between then (2004) and now (2018) in the context of the Java PathFinder tool, its symbolic execution component that is now called Symbolic PathFinder, and closely related approaches that target testing of software that manipulates complex data structures. We close with directions for future work.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Software testing and debugging*;

KEYWORDS

Software Testing, Symbolic Execution, Model Checking

ACM Reference Format:
Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2018. Test Input Generation with Java PathFinder: Then and Now (Invited Talk Abstract). In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3213846.3234687

OVERVIEW

Fourteen years ago, at ISSTA 2004, we published a paper [17] that described three automated testing techniques and applied them to systematically create structurally complex test inputs for high code coverage of an intricate program. Back then, model checking [2] was already established as the premier method for *hardware* verification, but was only starting to find its impact on validating *software* systems. The field of *software model checking* was in its early stages and researchers were starting to realize its usefulness, not just for traditional verification tasks, but even more so for *systematic bug finding*, specifically to find subtle bugs that are hard to find otherwise [6, 17]. At the same time, constraint solvers were coming of age. Propositional satisfiability (SAT) solvers were already handling industrial scale benchmarks [5, 13]. Satisfiability modulo theories (SMT) solvers were starting to gain traction as a practical backend tool [15]. Overall, the field of software analysis was primed for significant advances to be fueled by modern solvers.

The initial techniques for software model checking, which appeared in the mid and late 1990's focused on properties of *control*, where data was abstracted away or concretely provided, often via non-deterministic choice operators [4, 6–8, 16]. The early 2000's saw the beginning of systematic techniques that focused instead on properties of *data*, specifically heap-allocated data structures that have complex structural integrity constraints and provided constraint-based systematic test generation using SAT and dedicated solvers [1, 12].

In 2003 [9], we introduced the first software model checking approach that could, in principle, systematically test programs for *both* properties of data, which were handled by a generalization of traditional symbolic execution [3, 10], and properties of control, which were handled by embodying the approach in the Java PathFinder (JPF) model checker [16] that natively handled them using its custom Java Virtual Machine (JVM). Our ISSTA 2004 paper built on that approach to provide automated generation of test suites at the black-box level and at the white-box level, and supported the use of specifications given as executable code, e.g., method preconditions written as Java predicates [11]. In addition, our techniques provided testing in the traditional spirit of executing unit tests using standard (concrete) execution, and testing based on symbolic execution, where *path conditions* are built and solved (if feasible) to analyze the program's execution paths. Our work employed the Omega Library decision procedure for solving path conditions [14, 18].

In this talk, we take a look back at the systematic test input generation problem, specifically in the context of JPF and generalized symbolic execution, and provide an introductory account of the foundational ideas and our basic techniques. Our goal is not to provide a formal treatment or a critical evaluation of the work or its impact, rather to present the basic concepts and how they fit in the

ACM ISBN 978-1-4503-5699-2/18/07.
https://doi.org/10.1145/3213846.3234687

context of specification-based black-box and white-box testing in a way that is accessible to a wide audience, including students who may want to learn more about the exciting area of automated test input generation. Moreover, we do not intend to review the vast amount of literature in this area, rather we focus on some some key related projects that we were a part of and some very closely related work.

ACKNOWLEDGMENTS

This material is based upon work partially supported by the US NSF under Grant Nos. CCF-1319688 and CCF-1718903. We thank Darko Marinov for insightful comments.

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *ISSTA*.
[2] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.
[3] L. A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2, 3 (May 1976).
[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. 2000. Bandera: Extracting Finite-state Models from Java Source Code. In *ICSE*.
[5] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *SAT*.
[6] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *POPL*.
[7] Klaus Havelund and Jens U. Skakkebæk. 1999. Applying Model Checking in Java Verification. In *SPIN*.
[8] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997).
[9] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*.
[10] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19, 7 (1976).
[11] Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* (1st ed.). Addison-Wesley.
[12] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*.
[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC*.
[14] William Pugh. 1992. A Practical Algorithm for Exact Array Dependence Analysis. *Commun. ACM* 35, 8 (1992).
[15] Cesare Tinelli. 2002. A DPLL-Based Calculus for Ground Satisfiability Modulo Theories. In *JELIA*, Vol. 2424.
[16] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. 2000. Model Checking Programs. In *ASE*.
[17] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *ISSTA*.
[18] The Omega Project Website. 2018. https://www.cs.umd.edu/projects/omega/. (2018).