JPF-Doop: Combining Concolic and Random Testing for Java

[Extended Abstract]

Marko Dimjašević
School of Computing, University of Utah, USA
marko@cs.utah.edu

Zvonimir Rakamarić
School of Computing, University of Utah, USA
zvonimir@cs.utah.edu

ABSTRACT

Achieving high code coverage during software testing is important because it gives a measure of how thoroughly the software has been tested. However, reaching high code coverage in testing of real-world software is challenging due to its size and complexity. Our paper addresses this challenge by proposing an automatic multipronged approach. In particular, we propose an iterative algorithm for generating unit tests that meaningfully combines concolic execution and random testing. The algorithm aims to exploit the advantages of both random testing and systematic software verification techniques. We implemented the algorithm by integrating a Java Pathfinder's concolic execution engine and Randoop, and dubbed the implementation JPF-Doop. Preliminary experimental results show that JPF-Doop outperforms Randoop in terms of code coverage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification, Experimentation, Performance

Keywords

Concolic testing, Java Pathfinder, Random testing

1. INTRODUCTION

Software has become very complex nowadays, and is comprised of many libraries providing various underlying functionalities. Software libraries are also large and contain hundreds of thousands of lines of code. Because of their size and complexity, it is hard and unaffordable to check the correctness of libraries manually. To address these challenges, many automatic software testing tools and techniques exist today (e.g., OCAT [9], Agitator [1], Evacon [8], Seeker [14], DSD-Crasher [3]) and they are suc cessful to a certain degree. These techniques often do unit testing, that is, they test small parts of a big software library in isolation. Tools like PEX [15] test one program method at a time, while tools like Randoop [12] form random test drivers that are sequences of library method calls. Quality of such testing is often measured by various code coverage metrics. Because such testing techniques either focus on a single method at a time or just form random method call sequences, they often fail to drive program execution to hard-to-reach sites in software libraries, and as a result code coverage is suboptimal.

We have been working on addressing the aforementioned issues by combining two automated techniques into a novel algorithm. Our emphasis is on employing *concolic* (symbolic + concrete) execution [13, 6] and feedback-directed random testing [12] in software testing. To evaluate our algorithm, we have implemented it by integrating jDART [5, 7], a Java Pathfinder's component for concolic execution, and Randoop, a random testing engine that is feedback-directed. We dubbed the integration JPF-Doop, thereby acknowledging both tools. Furthermore, we have used code coverage metrics to compare JPF-Doop and Randoop on several real-world complex Java libraries. The results show that JPF-Doop typically outperforms Randoop on very complex libraries.

2. ALGORITHM

First, we give an overview of both concolic execution and feedback-directed random testing. At the end of the section, we describe our iterative automatic algorithm that combines these two techniques.

2.1 Concolic Execution

Concolic execution [13, 6] is a formal technique for executing a program, in which symbolic [11] and concrete execution are interleaved. The execution starts from the beginning of the program with concrete input parameter values. The execution is directed by the concrete input values. As the execution proceeds, symbolic path constraints, which are logic formulae, are generated for that particular execution. After the execution terminates, a sub-formula of the path constraint that represents a conditional statement is negated, and a new vector of concrete input values is generated using a constraint solver. This new vector is used for another execution that will follow a different path, the path defined by the path constraint with the negated conditional statement. These steps are repeated until all possible symbolic path

Package	Classes	Methods	Instructions	Branches
Collections (org.apache.commons.collections)	422	3894	58470	6244
Primitives (org.apache.commons.collections.primitives)	231	1858	18490	1446
Codec (org.apache.commons.codec)	77	594	24884	1829
Math (org.apache.commons.math3)	845	6886	288250	18576
Net (org.apache.commons.net)	161	1637	32194	2734

Table 1: Packages from the Apache Commons library used in our experiments.

constraints (i.e., all possible paths) have been considered, or until a given time limit is reached.

The main advantage of concolic over concrete execution is that it significantly reduces a search space over input parameters, i.e., it can potentially avoid the search space explosion problem because one symbolic execution potentially covers a multitude of concrete executions. On the other hand, concolic execution also has an advantage over traditional symbolic execution because when a conditional cannot be resolved symbolically, it is replaced by the concrete value from the concrete execution. This replacement reduces the abstraction level of the symbolic execution, but at the same time allows for execution to continue where the symbolic execution was not able to proceed on its own.

2.2 Feedback-Directed Random Testing

Instead of executing a program with concrete input values repeatedly and in a completely random fashion when performing testing, some information from earlier executions can be used to direct new executions. In this way only unit tests that execute all of their method calls can be generated, unlike unit tests that have totally random method sequences that would have been generated otherwise, and that would terminate their execution prematurely due to an invalid method sequence prefix.

Randoop [12] uses information from previous executions to direct further executions. The tool keeps two types of sets of method invocation sequences — valid and invalid, i.e., those that do not violate any property, and those that do, respectively. Both sets are empty in the beginning. The most basic property is a program termination without any error or exception thrown. Randoop starts by selecting a public method and an existing method sequence from the set of valid sequences, both selected at random. Then it stitches the selected method call to the end of the method sequence, and asserts that the new extended sequence is neither in the valid nor in the invalid set. It executes the new extended sequence, and checks for property violations. If the new extended sequence does not violate any property, the sequence is added to the valid set; it is added to the invalid set otherwise. Randoop keeps on repeating extending valid sequences until a time limit has been reached.

2.3 Iterative Algorithm

Figure 1 shows our proposed algorithm. The algorithm combines concolic execution and feedback-directed random testing into an iterative algorithm [4]. Random testing is used for global search by generating random test drivers. Local search is achieved by turning concrete input values in test drivers into symbolic values and performing concolic execution on such modified drivers. Because of the global search, testing is not getting stuck in one local area of the search space. On the other hand, concolic execution enables local exploration of the search space in the neighborhood of a test driver.

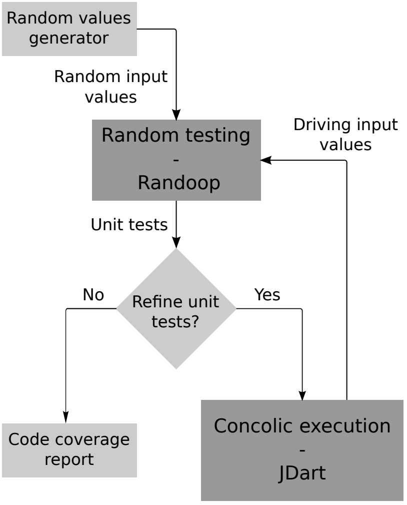

Figure 1: Our iterative algorithm for unit test generation. The algorithm combines concolic execution and feedback-directed random testing.

The algorithm generates unit tests that are test drivers. The test drivers navigate program execution through multiple method calls to hard-to-reach code sites. The algorithm iterates the unit test generation over several steps. The first step of the algorithm consists of feedback-directed random testing that generates test drivers for a software library. The second step performs systematic concolic execution of the appropriately modified test drivers generated in the first step. The concrete values generated in the second step are passed on to the third step, where they are used for another run of random testing. Furthermore, these steps can be repeated in a loop to iteratively generate unit tests that reach increasingly more points in the search space, as illustrated in Figure 1. Code coverage is measured in the end.

3. IMPLEMENTATION AND EVALUATION

In order to evaluate the algorithm described in Section 2, we implemented it in a tool named JPF-Doop. Furthermore, we evaluated the algorithm by running experiments with JPF-Doop against various real-world complex Java libraries. At the end of this section we report our experimental results.

3.1 JPF-Doop

The implementation glues together jDART and Randoop, and it is written in the Python programming language. This

Package/Tool	Randoop			JPF-Doop		
	Instructions	Branches	Unit tests	Instructions	Branches	Unit tests
Collections	50.94% ± 0.99%	38.96% ± 1.09%	20886.7 ± 3121.1	40.56% ± 2.51%	28.31% ± 2.67%	5344.4 ± 1827.7
Primitives	58.55% ± 0.05%	61.56% ± 0.07%	56900.1 ± 2192.4	58.13% ± 0.40%	61.20% ± 0.29%	45653.4 ± 1561.6
Codec	84.19% ± 0.17%	68.84% ± 0.38%	35755.1 ± 753.0	84.24% ± 0.28%	68.55% ± 0.57%	31340.9 ± 691.4
Math	15.57% ± 0.12%	2.14% ± 0.06%	77.5 ± 1.4	21.22% ± 5.87%	6.57% ± 5.15%	1651.6 ± 591.4
Net	31.41% ± 4.08%	14.06% ± 2.80%	1219.3 ± 358.4	42.05% ± 2.04%	23.41% ± 2.34%	13473.2 ± 5966.0

Table 2: Summary of our experimental results. Instruction (branch) coverage is given relative to the total number of instructions (branches) in the package.

is carried out using Python scripts that set up an environment for Randoop and jDART, and facilitate interaction of these two otherwise independent tools. In this environment Randoop and jDART make a loop of information flow, where output that Randoop generates is used to form input for jDART, and conversely, output that jDART generates is used to form input for Randoop. Note that in the first execution of Randoop there is no output from jDART to be provided to Randoop; hence, completely random inputs are used initially.

JPF-Doop first executes Randoop, and collects JUnit unit tests that Randoop has generated. Randoop generally generates a lot of unit tests, so JPF-Doop makes a selection of a small subset of unit tests at random. It proceeds by modifying the selected unit tests in such a way that concrete input parameter values are replaced by symbolic values, that is, by symbolic variables. These modified unit tests are suitable for concolic execution through exploration of all possible paths such symbolic variables enable.

The next step JPF-Doop performs is invocation of jDART on each modified unit test in order to carry out concolic execution. jDART was adapted so that in the end of the concolic execution process it writes to a file all representative concrete values of the symbolic variables generated during the execution. JPF-Doop scrapes up these representative concrete values and forms input for the next execution of Randoop. The collected values are used as a pool of input parameter values for all methods in all classes that Randoop will test, thus enabling more valid unit tests to be generated. By default, Randoop has a pool of standard input values that it uses in the process of unit test generation, but this pool is quite limited. The pool of concrete values generated by jDART enables Randoop to reach much deeper program states that would otherwise be virtually unreachable by random testing.

3.2 Experimental Results

To evaluate the effectiveness of JPF-Doop, we selected five package from the Apache Commons library, which are summarized in Table 1. In total, the packages contain 422288 source code instructions and 30829 branches. The packages include container classes (Collections and Primitives), complex computation classes (Codec and Math), and classes that interact with the environment (Net).

In our experiments we measured two code coverage metrics — instruction coverage and branch coverage. Since both Randoop and JPF-Doop are based on randomized algorithms, we executed each tool 10 times on each of the packages, with a time limit of five minutes per execution. Table 2 summarizes our experimental results by providing the averages and standard deviations for both instruction and branch coverage. The results are calculated relative to the total number of instructions (branches) in a package. The summaries are given in form $m \pm \sigma$, where m is the average and σ is the standard deviation. The table also includes the total number of generated unit tests for each package.

As it can be seen from Table 2, in the given time limit Randoop achieved about 10% higher instruction and branch coverage than JPF-Doop for the Collections package. On the other hand, JPF-Doop achieved about 10% higher instruction and branch coverage than Randoop for the Net package, which includes interaction with a network environment. For the Math package, which contains a lot of complex branching, improvement by JPF-Doop compared to Randoop is not as significant, but it is noticeable. Note that JPF-Doop has a large deviation there because in two executions it achieved much higher coverage. There is no significant difference in terms of coverages provided by Randoop and JPF-Doop for the Primitives and Codec packages.

Unit tests generated by each execution of Randoop — in Randoop itself or when Randoop is used as a part of the JPF-Doop tool — are such that no unit test is subsumed by another generated test. In other words, if unit test A is a sub-sequence of instructions in unit test B, then unit test A is discarded while unit test B is kept. From Table 2 it can be concluded that the more unit tests a tool was able to generate, the better instruction and branch coverage it was able to provide.

4. CONCLUSIONS AND FUTURE WORK

Because the process of designing and writing software systems is inevitably erroneous, it is important to have automatized ways of finding errors in such systems. Our work on JPF-Doop addresses the issue of achieving high code coverage in software testing using automatic techniques, thus increasing the likelihood of finding errors in software. We compared our automatic iterative algorithm against feedback-directed random testing, which has been shown to generally provide high code coverage, and demonstrated improvements.

JPF-Doop is a work in progress. We plan to combine automata learning and concolic execution with the goal of further improving automatic testing of software libraries. In order to improve code coverage in testing of large-scale libraries, we will build on our current work. MACE [2] is a tool that combines automata learning and concolic execution. Unlike MACE, our future work will focus on large-scale libraries instead of implementations of network protocols, and with the imperative of being completely automatic.

5. ACKNOWLEDGMENTS

This research was sponsored by the United States National Aeronautics and Space Administration (NASA) under Prime Contract No. NNA10DE60C and by the Google Summer of Code 2013 program.

6. REFERENCES

[1] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, page 169–180, New York, NY, USA, 2006. ACM.

[2] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. J. Wu, and D. Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security*, volume 11, 2011.

[3] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: a hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[4] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *ICSE 2013*, pages 132–141, 2013.

[5] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In A. Miné and D. Schmidt, editors, *Static Analysis*, number 7460 in Lecture Notes in Computer Science, pages 248–264. Springer Berlin Heidelberg, Jan. 2012.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[7] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 268–279. ACM, 2013.

[8] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008*, pages 297–306, Sept. 2008.

[9] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, page 159–170, New York, NY, USA, 2010. ACM.

[10] Java path finder. http://babelfish.arc.nasa.gov/trac/jpf.

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

[12] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 75–84, May 2007.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[14] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. *SIGPLAN Not.*, 46(10):189–206, Oct. 2011.

[15] N. Tillmann and J. d. Halleux. Pex–White box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, number 4966 in Lecture Notes in Computer Science, pages 134–153. Springer Berlin Heidelberg, Jan. 2008.