Automatic Testing of Software Libraries

Marko Dimjašević
School of Computing
University of Utah
Salt Lake City, Utah, USA
Email: marko@cs.utah.edu

***Abstract*—Achieving high code coverage during software testing is important because code coverage 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. This research addresses the challenge of reaching high code coverage by proposing two automatic approaches. One is an iterative algorithm for generating unit tests with concolic execution and random testing. The algorithm exploits both random testing and systematic software verification techniques. Preliminary results show the algorithm outperforms existing techniques. The other approach, which is our future work, uses automata learning to learn a component interface and to generate unit tests based on the learned interface.**

I. Introduction

Software has become very complex 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, and they are successful 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 [1] test one program method at a time, while tools like Randoop [2] 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 focus on a single method at a time, or because they 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 various automated techniques from different computer science fields in novel ways. Our emphasis is on employing automata learning [3] and *concolic* (symbolic + concrete) execution [4] [5] in software testing. Automata learning and concolic execution, when combined, have strong advantages in their application in software testing.

II. Background

A. Automata Learning

Automata learning is an algorithm for learning an unknown regular language by examples. A learning process starts with a learner forming and asking membership queries — *Is this word in the language?* — and a teacher that knows the language answering *yes* or *no* depending on if the word is in the language. Once the learner is confident it has learned the language, it forms a conjecture about the language it is learning and asks the teacher if that is the correct language. The teacher either confirms in the case of the correct conjecture or provides a counterexample otherwise.

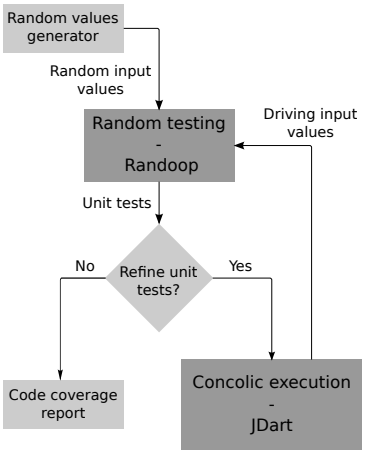

Fig. 1. Iterative algorithm for unit test generation. The algorithm combines concolic execution and feedback-directed random testing.

We propose to use automata learning in the context of software testing. Symbols of a regular language are method names with their input parameters. A library consists of numerous methods. Therefore, to learn the language means to learn sequences of legal calls to the library methods. The learner is an algorithm that searches for legal sequences, and the teacher is an execution that either finishes regularly, or finishes in an error. Once the learner learns the language, an automaton representing the language is built. The automaton corresponds to the interface of the library. In order to test the library, unit tests are generated from the learned interface, and code coverage is measured in the end.

B. Concolic Execution

Concolic execution is a formal technique for executing a program in which symbolic [6] 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 subformula of the path constraint that represents a conditional statement is negated and a new vector of 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 constraints, that is all possible paths, have been considered, or until a given time limit is reached.

The main advantage of concolic execution over concrete execution is that it significantly reduces a search space, i.e., it can potentially avoid the search space explosion problem because one symbolic execution potentially covers multitude of concrete executions. On the other hand, concolic execution has an advantage over traditional symbolic execution because when a conditional statement cannot be resolved symbolically, it is replaced by the concrete value from the concrete execution. This replacing reduces the abstract 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.

III. Current Research

Our current research is geared towards improving code coverage in software testing by combining concolic execution and random testing into a novel iterative automatic algorithm. Garg et al. [7] have a recent publication with similar work, and we believe the combination is the first step towards combining automata learning with concolic execution. Random testing is used for global search by generating random test drivers. Local search is achieved by turning concrete input values in the test drivers into symbolic values and performing concolic execution on such modified drivers. Because of the global search, testing is not 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 shows our algorithm. The algorithm generates unit tests that are test drivers. The test drivers navigate program execution through multiple method calls to the hard-to-reach sites. The algorithm iterates the unit test generation over a few steps. The first step of the algorithm consists of feedback-directed random testing that generates test drivers of a software library. The second step performs systematic concolic execution of the 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 the random testing. Code coverage is measured in the end. Furthermore, these steps can be repeated in a loop to iteratively generate unit tests that reach increasingly more sites of the search space.

A. Preliminary Results

The algorithm has been implemented by combining Java PathFinder's JDart [8] component and Randoop into a new software tool dubbed J-Doop. To evaluate the tool, we compared its results to those of Randoop for several Java libraries.

Library	LOC	Tool	Instructions	Branches
CEV	282	Randoop	79%	72%
		J-Doop	90%	81%
MapUtils	537	Randoop	63%	69%
		J-Doop	63%	69%
EWAH	967	Randoop	79%	71%
		J-Doop	84%	77%

TABLE I
COMPARISON OF CODE COVERAGE OBTAINED BY RANDOOP AND J-DOOP.

The experiments were run on a laptop with Intel Core i5 with the clock of 2.50 GHz and 4 GB of DDR3 RAM memory.

Table I shows our preliminary results. Instructions coverage and branches coverage are reported, where each library was tested for one hour. Crew Exploration Vehicle (CEV) is a model of flight phases for a NASA's space-craft. The model features a lot of branching. MapUtils is a collection class from the Apache Commons library. EWAHCompressedBitmap (EWAH) is a class from the JavaEWAH, a compressed alternative to the Java's BitSet class.

IV. Future Work

We are planning to combine automata learning and concolic execution with the goal of 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 [9] 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.

V. Conclusion

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 addresses the issue of achieving high code coverage in software testing with 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 promising improvements.

References

[1] N. Tillmann and J. d. Halleux, "Pex–White box test generation for .NET," in *Tests and Proofs*, pp. 134–153, Springer Berlin Heidelberg, 2008.

[2] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007*, pp. 75–84, 2007.

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE-13*, p. 263–272, 2005.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI 2005*, p. 213–223, 2005.

[6] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, 1976.

[7] 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*, pp. 132–141, 2013.

[8] D. Giannakopoulou, Z. Rakamarić, and V. Raman, "Symbolic learning of component interfaces," in *SAS 2012*, pp. 248–264, 2012.

[9] 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*, vol. 11, 2011.