Using Coverage Analysis to Automatically Generate Test Cases

Tim A. Majchrzak
Department of Information Systems
University of Muenster
Muenster, Germany
Email: tima@wi.uni-muenster.de

1. Extended Abstract

Unit testing is an important part of the overall test process as it offers the chance to discover a substantial part of errors in early development phases. It is hard to write unit tests that guarantee the desired coverage of the code. Ideally, they could be created automatically with no or little intervention by testers. We hence investigate on this topic.

Our tool, Muggl (**Mu**enster **g**enerator of **gl**ass-box test cases), is based on GlassTT [1] which was designed at our department. Due to changes of fundamental design principles we have rewritten Muggl from scratch. It only incorporates the constraint solver built for GlassTT which has proven to be powerful. With the design changes made, we aim at learning from problems encountered with symbolic execution, while keeping the already known amenities.

Muggl symbolically executes class files containing bytecode, as for example generated by the Java compiler `javac`. Using byte code instead of source code has two main advantages. Firstly, optimizations done by the compiler are taken into account. And secondly, many languages can be compiled to Java bytecode and therefore tested using just one tool. Muggl uses a symbolic implementation of the Java virtual machine (*JVM*) [2] that offers the ability to treat input parameters of a method as logic variables. Conditional jumps and other instructions that allow alterations of the control flow lead to *choice points* generated when executed.

Using a search algorithm processing the tree of potential paths through the program, Muggl tries to determine sets of parameters for the method to be tested. Each set of parameters found and the corresponding result establish a test case. Since the number of paths through a program is typically infinite, we need means to select a small finite set of representative test cases. This selection is done based on *control-* and *data-flow coverage*. Covering control-flow is a possible goal of unit testing. However, experiments show that it is not sufficient in practice and that many errors are not exposed based on this criterion. Hence, we suggest tracking the data-flow as well to check the way data is loaded and changed. Due to a number of reasons we have decided to statically generate *control-flow graphs* and *definition-usage (DU) chains* rather than generating them only-the-fly. This significantly speeds up elimination of test cases.

Generating the control-flow graph is very straightforward. Bytecode instructions form one or more edges in the graph according to their execution characteristics. Conditional jumps e.g. have two edges. Also taking exceptions into consideration is very important, since exceptions are a fundamental part of Java and Java bytecode. Intra- and interprocedurally tracking def-use chains is far more complex. We have developed a novel algorithm to generate those chains. Outlining it in this abstract is however out of scope.

Based on coverage data we eliminate test cases. When Muggl finishes execution, a number of test cases larger than the optimal number is left. Reducing their number based on their contribution to the overall coverage of control-flow edges and DU chains is a *NP-complete set-coverage problem*. We have implemented a *greedy algorithm* that very well approximates the optimal result. We then tested the sketched approach with a number of simple and some larger programs. Elimination of test cases has proven to be very fast and to eliminate (almost) any redundant test case. Execution times usually did not exceed fractions of a second. Even eliminating 374.879 test cases only took about 3,5 s.

2. Presentation Focus

Presenting our approach, we would like to only shortly introduce into Muggl's basics. We rather aim at discussing using coverage to eliminate test cases. This offers the chance to also discuss whether coverage alone can be used to judge if all test cases required to successfully test a program have been found. Moreover, we would like to present some experimental results. Outlining future plans and general ideas would be an addition. Receiving feedback that helps to reconsider ideas and to prioritize work would be perfect.

References

[1] R. A. Mueller, C. Lembeck, and H. Kuchen, "Generating glass-box test cases using a symbolic virtual machine," in *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004)*, 2004.

[2] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.