CS/SE 6367 Course Project Proposal

ASM Bytecode Automated Coverage Collection

Hung-Jui Guo
The Software Engineering Department, The University of Texas at Dallas
Richardson, Texas
hxg190003@utdallas.edu

James Chen
The Software Engineering Department, The University of Texas at Dallas
Richardson, Texas
jnc190001@utdallas.edu

Ximeng Liu
The Computer Science Department, The University of Texas at Dallas
Richardson, Texas
xxl176530@utdallas.com

Meng-Hsiang Chang
The Software Engineering Department, The University of Texas at Dallas
Richardson, Texas
mxc176430@utdallas.edu

ABSTRACT

Program is making a very important role in human society. People's civilization relies on programs' functions. Every phone call, every airline route, they all rely on the precise performance of the program running. Delivering the correct functionality and eliminating potential bugs is the optimal goal for every software engineer.

Every program has to be tested thousands of times, and there are many different test methods, which includes Unit Testing; Integration Testing; System Testing and Acceptance Testing. Through all the testing, Unit Testing is the most basic and the most frequently used testing method. There is a report that says 65% of bugs in the program can be found through Unit Testing. To perform Unit Testing, how to measure the testing quality is based on code coverage test. The code coverage is dominated by Statement Coverage in Control Flow Testing method. The importance of Statement Coverage and how to implement Statement Coverage is the main focus in this article.

KEYWORDS

JUnit, ASM, Java, Java agent, Maven

ACM Reference Format:
Hung-Jui Guo, James Chen, Ximeng Liu, and Meng-Hsiang Chang. 2020. CS/SE 6367 Course Project Proposal: ASM Bytecode Automated Coverage Collection. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEMS TO BE SOLVED

Statement Coverage is under White Box testing, it will determine the program whether it has bugs based on examining if every statement has been executed. But there are some concerns, does statement coverage working functional on every program? Does Statement Coverage provide enough code coverage? How is the Statement Coverage efficiency compared to other code coverage methods? How is the Statement Coverage performance in modern industry uses?

2 BASIC TECHNIQUES

2.1 Java Agent

The code instrumentation way that we learnt in the class has to modify the bytecode file on the file system. There are several limitations for that. First, the time can be wasted when reading and writing back the bytecode files from/to file systems. Second, the bytecode file on the file systems may be changed into a mess. Therefore, Java Agent [1] was proposed to change the bytecode file only in the memory during the JVM load time. Whenever, a Java bytecode file is loaded into JVM (for execution), Java Agent will transform that file and add instrumentations in the memory, leaving the actual bytecode file on file systems unchanged. There is a much cleaner and faster way to do code instrumentation. Java Agent can be directly combined with ASM to do bytecode instrumentation on-the-fly [5].

2.2 JUnit Listener

Results show that the quality of real world unit tests is low, and that in many cases, unit tests don't follow the well-known recommendations for writing unit tests. These early results demonstrate the need for more tools and techniques for refactoring of tests[7].

JUnit Listener In order to record the coverage information for each JUnit test method, you need to capture the start and end events for each JUnit test method, so that you can start recording when the test starts and stop recording when the test ends. JUnit RunListener provides a direct way to do that. There are plenty of example online [4].

2.3 Integration with Maven

Traditional software updating and maintenance are generally free to the software development process[8]. Based on Maven, a sophisticated software build tool, this paper presents an automatic update method: using POM model of Maven in user terminal management and update all dependent modules of the software, thus completing the entire software updates. This method makes the software update be incorporated into the Maven project management processes, enabling seamless continuous, automatic software build and update from source code changes to the terminal user. Experiments show that this method is feasible, effective.

Maven [6] is one of the most widely used build system for Java. It enables the users to build Java projects easily without worrying about the 3rd-party libraries, since the Maven build system will automatically download the 3rd-party libraries based on the Maven configuration file (pom.xml). The "mvn test" command is used to execute all the JUnit tests for a Maven project (Note that all the projects in the course project list appendix use Maven projects, and you can use that command to execute the tests that come with the projects). When firing the "mvn test" command, the Maven surefire plugin is used to find and execute all the JUnit tests. You can easily configure your Java Agent and JUnit listener for the surefire plugin in the Maven pom.xml file. Example for integrating Maven with both Java Agent and JUnit listener can be found online [3].

3 IMPLEMENTATION PLAN

Our implementation plan starts at 2/17 and ends at 5/1.

Project proposal: 2/17 - 2/21
Paper survey: 2/22 - 2/28
First phase implement: 2/29 - 3/13
Improve first phase: 3/14 - 3/28
Test and improve tool: 3/29 - 4/12
Evaluation and second phase document: 4/13 - 5/1

4 EVALUATION

Use ASM byte-code manipulation framework [1] to build an automated coverage collection tool that can capture the statement coverage for the program under test. Then, apply your tool to 10 real-world Java projects (>1000 lines of code) with JUnit tests (>50 tests) from GitHub [2] to collect the statement coverage for its JUnit tests. For statement coverage quality estimation score, we calculate the statement coverage point X, we will compare X with the total statement number Y. The result score will be: S = X/Y. We can use this final score as our estimation of the quality of the real-world Java projects' JUnit tests. If S1 is higher than S2, it represent that the first Java project has better JUnit tests than the second Java project.

REFERENCES

[1] [n.d.]. *ASM Official Website.* Retrieved February 19, 2020 from https://asm.ow2.io/
[2] [n.d.]. *Github.* Retrieved February 19, 2020 from https://github.com//
[3] [n.d.]. *How to Configure JUnit Listeners.* Retrieved February 19, 2020 from https://asm.ow2.io/
[4] [n.d.]. *How to implement RunListener a JUnit Listener example.* Retrieved February 19, 2020 from https://memorynotfound.com/add-junit-listener-example/
[5] [n.d.]. *An introduction to Java Agent and bytecode manipulation.* Retrieved February 19, 2020 from https://www.tomsquest.com/blog/2014/01/intro-java-agent-and-bytecode-manipulation/
[6] [n.d.]. *Maven in 5 Minutes.* Retrieved February 19, 2020 from https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html/
[7] Dor Ma'ayan. 2018. The quality of junit tests: an empirical study report. In *2018 IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE).* IEEE, 33–36.
[8] Xiong Zhen-hai and Yang Yong-zhi. 2014. Automatic updating method based on maven. In *2014 9th International Conference on Computer Science & Education.* IEEE, 1074–1077.