Coverage Collection and Test Generation in Java

Final Report

Wilfred Labue
Computer Science Department
University of Texas at Dallas
Richardson, TX 75080
wxl180015@utdallas.edu

Lathwahandi De Silva
Computer Science Department
University of Texas at Dallas
Richardson, TX 75080
ldd170230@utdallas.edu

Ning Kevin Chen
Computer Science Department
University of Texas at Dallas
Richardson, TX 75080
nkc160130@utdallas.edu

ABSTRACT

In this project we will be using 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 we will apply our tool to 10 real-world Java projects which contains at least 1000 lines of code with JUnit tests at least 50 tests collected from GitHub [2] to collect the statement coverage for its JUnit tests.

The tool will be a Java based agent with ability to perform on-the-fly code instrumentation, and it will be able to store the coverage for each test method in the file system. The tool will be integrated via the Mavin build system which will in turn will simplify the tool execution by reverting the execution command to “mvn test”. In order to achieve this integration with the Maven build system we will create the necessary changes in the pom.xml file of the project located in test.

1. Coverage Collection

The primary objective of phase one is providing an automated coverage tool that captures code coverage, specifically statement coverage in a program which is subjected to various test cases. In this section we will be examining some primary objectives that should be achieved in phase one.

1.1 Statement Coverage

The premises of the coverage collection tool is to collect statement coverage of program subjected to a test. Statement coverage is a technique that is used to design white box test cases. This technique involves execution of all statements of the source code at least once. This methodology is used to calculate the total number of executed statements in the source code against the total statements present in the source code.

2. Test Generation Techniques

Since the project is in preliminary stage we decided to expand the initial scope to garner a better understanding of finding the optimal techniques we could implement to refine the solution. In the following subsection we will be discussing the various techniques we are considering to implement on the solution.

2.1 Pairwise Testing

Pairwise testing is a black-box testing technique. That is, it does not rely on modifying or manipulating the code to generate test cases. Instead, it relies on passing in different values into parameters to the functions within the program to generate tests. Now, say you have 34 checkboxes on a website. Since each checkbox can be set to checked or unchecked, the number of total inputs is 2^{34} inputs, or $1.7 * 10^{10}$ inputs. That means that, in order to test all inputs individually, you need to generate $1.7 * 10^{10}$ tests! Now, pairwise testing makes this much easier. If you consider that there are 3-way or 4-way interactions between the checkboxes, you can bring the number of tests down to 33 or 85, respectively. That’s a lot more efficient than 2^{34} test cases! Usually, the more way-interaction a pair is, the more percentage of bugs it will catch. This method finds useful applications from pizza ordering websites to Traffic Collision Avoidance System modules.

2.2 Random Test Generation

This is a black box software testing technique where programs are tested by generating random independent inputs. In random testing a random number generator (e.g monkeys) is used to generate test cases and derive estimates for the reliability of the software using some probabilistic analysis. With random testing random inputs are entered into a system to see what the results are. A common example is the use of random integers to test a software function that returns results based on those integers. Other type of random testing may involve the use of heuristic , which guide the use of random inputs.

3. Code Coverage Tool Design

3.1 JUnit

JUnit is a unit testing framework used in Java programming. JUnit has been important in the development of test-driven development. It is a simple framework used to write repeatable tests and an instance of the xUnit architecture for unit testing framework. JUnit promotes the idea of “first test then code”, which emphasizes on setting up the test data for a piece of code that can be tested and then implemented.

3.2 Maven

Maven is a build automation tool used in Java projects. It automatically downloads Java libraries, as well as Maven plugins, from the Maven repository. Maven utilizes one file, the pom.xml, to configure the Maven project for building. POM stands for Project Object Model, and the file is in XML format. For example, one can use Spring Boot, a framework that allows you to build web applications using Java, in a Maven project..

3.3 ASM Bytecode Framework

ASM is an all purpose Java bytecode manipulation and analysis framework. it can be used to modify existing classes or dynamically generate classes , directly in binary form. The ASM library is used to generate , transform and analyze compiled Java classes , represented as byte arrays. ASM provides tools to read, write and transform such byte arrays by using higher level concepts than bytes , such as numeric constants , strings , JAva identifiers, Java types and Java class structure elements.

3.4 Java Agent

When it comes to the modification of the bytecode located in a file, there are several limitations induced as a result of this phenomenon. Initial limitation is the excessive consumption of time and secondly the risk induced approach that can create disruption among the bytecode in the file system in a fundamental level. Therefore, a Java agent is used to transform the byte code files during JVM, this tactic leads less complexity in regards to unforeseen altercations that can occur to the actual byte code file located in the system.[2]

4. Evaluation/ Team Schedule

4.1. Code Coverage Tool Evaluation

In order to correctly evaluate the results presented by our code coverage tool, the primary course of action taken will be cross reference among results produced by a pre-existing code coverage tool. Currently we have couple pre-existing code coverage tools that we could implement due to the preliminary nature of the project we have yet to determine the final examination methodology we will be implementing. The implementation options are as follows. Emma and Jacoco.

Emma is is an open-source toolkit for measuring and reporting Java code coverage. EMMA distinguishes itself from other tools by going after a unique feature combination. [1] The reason for considering Emma as a possible candidate is primarily due to the instrument classes for coverage either offline (before they are loaded) or on the fly (using an instrumenting application classloader). [1] The Code coverage tool Jacoco was created as a spiritual successor for Emma and Cobertura. The reason for selecting Jacoco as a candidate is due to its support for EclEmma plugin, support for Mavin, and active development team . [2]

4.2 Team schedule

The current team schedule consist of weekly physical meetings on Friday between 1pm-3pm, The primary reason for selecting this time slot is due to mutual team member availability presented in the allocated time slot. Another note be regarded is due to the preliminary nature of this paper the time slot presented above is by no means permanent and it is subjected to change. Over the following weeks we will be fleshing out the details regarding the nature of the strategies we will implement in conjunction to overlay/ implementation of the custom code coverage,

test case generation, and pre-existing code coverage tool that will be implemented for cross reference/ validation.

5. Phase I Implementation

This section will be mostly discussing the phase I implementation process and the components we used meet the objective.

5.1 **JUnit Listener**

JUnit Listeners are classes which receive callbacks about test execution. These callbacks are executed when there is any event in test execution lifecycle like just before the test has started execution, just after the test has finished, when the test has failed or when the test have passed. In the code implementation we extend RunListener in project for us to respond to events during a test run.

In the implementation phase the JUnit Listener served as a basis to obtain the test class name and the test name method name via testStarted method, during this time we create an instance of the hash map that will be used to allocate the line number and the class name. The testFinished method in the JUnit Listener combines the String indicating Test class/Test method with the hash map containing line coverage/class name into a singular hashmap named testCases . Once all the tests are executed, the testRunFinished method append all the results located in the hashmap testCases into the text file “stmt-cov.txt” located in the root directory of the project under the test.

5.2 **Maven Integration**

In order to keep the complexity of integrating/testing the application in different systems we created the project with maven dependency. To be elaborate the once the coverage collection tool’s dependencies/ plugin has been properly configured in the test project pom file the test project can execute the customized Junit runner and collect statement coverage by simply typing the command “mvn test” in the root directory.

5.3 **Java Agent implementation**

We use java agent to inject and intercept code during runtime and also operate on classes before and after they are used. We implement two ASM classes in the java agent, that is, the ClassReader which reads Java Bytecode from a file and ClassWriter writes the Bytecode to a file.

5.4 **ClassTransformVisitor**

In the ClassTransformVistor we extend the ClassVisitor method. This class invokes method transform visitor to obtain line coverage/class name.

5.5 **MethodTransformVisitor**

This is where a huge portion of line coverage is induced. In order to trace all the line coverage during each line instance, method addCoveredLine from the CollectCoverage class is called to store the line numbers and associated class names.

5.6 **CollectCoverage**

In this class we implement the HashSet data structure to store all the coverage information and later on dump them into a file once the test ends. In order to prevent duplicate entries of line number we initialized lineCovered hashmap with IntSets.

As we mentioned earlier our coverage collector is able to get all the line number of the code executed and the class name via invocations made by our extended class of Method visitor. These information is stored in LinesCovered HashMap which we initialized.

5.7 POM (Project Object Model)

A POM is a unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. This file is used to specify the dependencies/plugins for our test projects. Furthermore, this file is used to customize the settings used to compile our coverage collector tool.

5.8 Results

Due to some constraints we were unable to cross reference the results obtained by our coverage tool against Coverage tool Emma, however we cross referenced our coverage collection tool results in conjunction to project “common-dbutils” against Prof. Zhang’s reference results for “common-dbutils”. The results were as follows the size of

the stmt-cov file created by our automated coverage. tool was 938 kb, and it covered 310 tests. We ran the coverage tool on 9 other projects obtained via professor Zhang's recommended list. The result for the coverage in those respective projects can be found in the statement coverage or test project root folder. For reference purposes below is a snippet indicating code coverage for "common-dbuilts" produced by our code coverage tool.

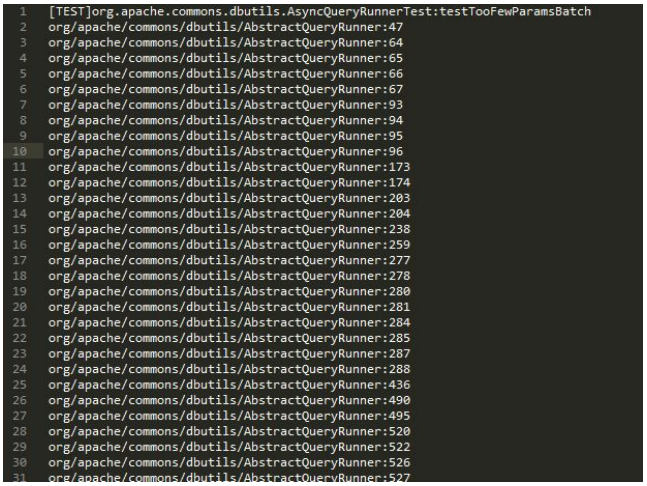

Figure: 1.1 Code Coverage results for Common-dbutils

6. Phase II Implementation

For this phase, we augment our coverage collection tool we implemented in phase one , to trace more information about the program's internal state. That is we trace accessible fields and variables for each method execution. These properties obtained via the trace can be used to deduce invariants of the respective data trace.

6.1 Method Transform

The primary means of obtaining the augmented coverage information was obtained in this premises. Here we override the method VisitVarInsc to obtainined the opcode and the respective index allocated to local variables. Upon obtaining the required information we deduced the variable value by identifying the type of the variable via the given Opcode. Upon deducing the index value and variable value we invoked the method addVariable located in the Collect Coverage.java to store the required information. Furthermore Visit Local Variable was used to deduce the name of the local variables given at its declaration.

6.2 Coverage Collection

The primary object of this section was to collect the augmented data and sort them in an acceptable manner for future use. In this component based on the data type we received from the opcode in the previous section we allocated the respective variable value and index value associated with the local variable into the hashmap "variable map". Another function located within the class specifically method "addName" was primarily responsible for obtaining the variable name and the associated index values via visitLocalVariable located in the method transformer. visitVarInsn calls the addVariable method, which simply collects the index and value of the variable in a HashMap.

6.3 JUnit Listener

This component was primarily responsible for appending the content collected during the instrumentation to a trace file. We produced two trace files for the sake of keeping every trace as readable as possible. During the second phase the two files "Invariant-trace" and "Name-trace" are produced to deduce the properties of the variables that was subjected to the instrumentation.

6.4 Evaluation and Invariant Candidacy

In this section we will be discussing about the results obtained via the data trace, overall structure, properties that can be deduced, how the said properties can be used to deduce invariants.

As we mentioned in the previous section we obtained two trace files via the augmentation tool besides the pre-existing line coverage trace file from phase I. In order to formulate an idea on we have obtained we will look at the two trace files separately.

Initial trace file subjected to examination is "Invariant-trace.txt"

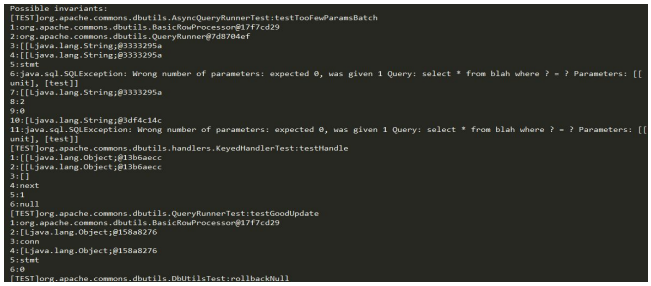

Figure 1.2 Invariant-trace coverage result for Common-dbuilts

The trace “invariant-trace.txt”, consisted following format,

At the beginning of each test method execution it will append the following test case information in the following format “**TestClassName:TestMethodName**”, and this was proceeded by producing variable information in the following format“**LocalVariableIndexNumber:LocalVariableValue”**. This can be used to deduce numerice properties about the local variable, however in order to gain a more sophisticated view of the internal state we produced the trace file subjected to examination which is “Name-trace.txt”.

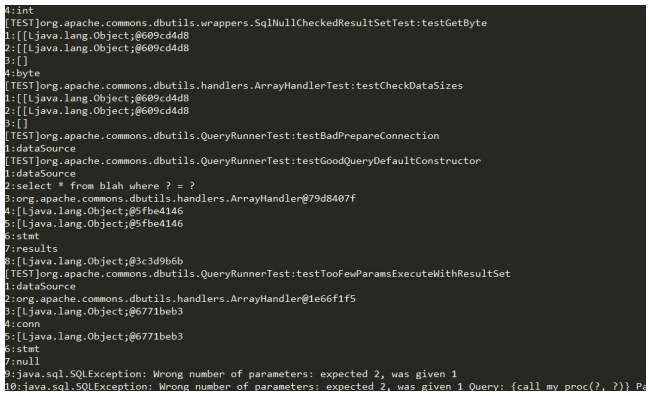

Figure 1.3 Name-trace coverage result for Common-dbuilts

The trace file “Name-trace” consisted of following format, just like it is predecessor “invariant-trace”, “Name-Trace” consist of test case information with the following format, **TestClassName:TestMethodName.** After this if there is an index value that consist of local variable declaration in the hashmap obtained via visitLocalVariable we outputted certain properties about the local variable in the following format , **IndexNumberofLocalVariable: LocalVariableName: TypeDescriptorOfLocalVariable: VariableValue.** In the event where we couldn’t find local variable declaration of the associated index values, we simply outputted the trace in the following format, **LocalVariableIndexNumber:LocalVariableValue.**

As far as invariant candidacy goes, the traced variables with the internal properties such as value, type can be used to determine/ verify invariants.

References

[1] Vlad Roubstov. 2006. a free Java code coverage tool. (January 2006). Retrieved September 30, 2019 from http://emma.sourceforge.net/

[2] Ling Ming Zhang. SE 4367 Course Projectl. Retrieved September 25, 2019 from https://elearning.utdallas.edu/bbcswebdav/pid-2735995-dt-content-rid-69989771_1/courses/2198-UTDAL-SE-4367-SEC501-84822/course-project.pdf

[3] About Stackify Stackify provides developer teams with unparalleled visibility and insight into application health and behavior. 2019. Code Coverage Tools: 25 Tools for Testing in C, C , Java. (May 2019). Retrieved September 30, 2019 from https://stackify.com/code-coverage-tools/