Implementing a Test Coverage Collection Tool

Ekansh Goyal
Dept. of Computer Science
University of Texas at Dallas
exg180024@utdallas.edu

Avani Sah
Dept. of Computer Science
University of Texas at Dallas
axs180221@utdallas.edu

Vinyas Raju
Dept. of Computer Science
University of Texas at Dallas
vxr170005@utdallas.edu

Aarya Reddy
Dept. of Computer Science
University of Texas at Dallas
axp170019@utdallas.edu

ABSTRACT

Software testing evaluates the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects/issues to ensure that the product is defect/issue free in order to produce the quality product. **Software Testing** Definition according to **ANSI/IEEE 1059 standard** – A process of analyzing a software item to detect the differences between existing and required conditions (i.e., defects) and to evaluate the features of the software item. Code coverage is the percentage of code that the automated tests cover. It just determines which statements in a code are being executed through a test run, and which statements are not. In general, a code coverage system collects information about the running program and then combines that with source information to generate a report on the test suite's code coverage. If a line of code or program has a higher test case coverage, it means that the source code executed during testing has a lower chance of containing software bugs or issues.

KEYWORDS

Statement Coverage, Test Coverage, Branch Coverage, Test Coverage Collection Tool, Software Testing.

1. The Problem Statement

We cannot easily depend on Unit tests as a metric for software quality as even after successful completion of the created unit test cases, a program does not guarantee that the code is bug free. We aim to capture the test coverage for numerous Junit test cases that are defined on existing projects to measure and evaluate the test suite's ability to cover every statement and branch to ensure the good quality of the project. The result or data we receive after this can be later used to create more effective software debugging.

2. Design of Our Technique

We are going to use the ASM framework to retrieve the statement coverage of the program under testing. ASM is an all-purpose Java bytecode manipulation framework which provides some common bytecode transformations and analysis algorithms from which custom complex transformation and code analysis tools can be built. We intend to use this framework to develop our own tool that could compute the statement coverages in Phase-1 of the project and extend the functionality to branch coverage in our project's Phase-2.

3. Our Implementation / Study Plan

The goal is to create an automated code coverage tool using ASM manipulation framework. The project is divided into two phases and the tasks for each phase are mentioned:

Phase - 1

- Build an automated tool to collect statement coverage of the programs that will be tested.
- Use the tool on at least 10 real world Java projects with more than 1000 LOCs and more than 50 Junit Tests selected from GitHub to find the statement coverages for all.

Specifications:

- Perform on-the-fly code instrumentation using Java Agent.
- Each statement coverage for each test method will be stored in file system.
- Tool can be triggered with a simple "mvn test" by integrating it with Maven build system and changing the pom.xml file of the project.

Phase - 2

- Use the basic Daikon technique via byte-code instrumentation to augment the functioning of the collection tool.

Specifications:

- Each variable value (local variable and accessible fields) for each method will be traced.
- The data collected will be traced to infer possible Single Variable and Single Numeric Variable invariants.

4. Phase I

4.1 Existing Techniques

Several code testing coverage tools are available as eclipse plugins that integrate with the JUnit testing framework. Moreover, they are available for public use. We discuss DJunit and Eclemma and here.

4.1.1 DJunit

Djunit is an eclipse plugin which performs the JUnit tests to calculate the code coverage for line, branch, package, file and overall program. It is an open source tool for java code coverage. This tool generates and shows directly the code coverage granularities of the JUnit tests that are performed in Eclipse IDE.

4.1.2 Eclemma

It is used to calculate the statement coverage metric. Its open source and used for code coverage. It highlights the fully covered code in green, partly covered code in yellow and uncovered code in red. In addition, it adds a coverage mode in Eclipse IDE as the existing modes like run and debug.

4.2 Implementation

The project phase 1 uses the Java Agent to create an object of MyClassTransform class. It implements the ClassFileTransformer. This object is passed to the addTransformer function of instrumentation class. This function registers this transformer which in turn will help in manipulation of the classes. Instrumentation class helps in coverage analysis by manipulating the classes.

The "transform" function of ClassFileTransformer is used to create objects for ClassReader, ClassVisitor, and ClassWriter. These in turn read the class and method details and keep track of visiting them and manipulates the bytecode.

All these classes collectively help in tracking the statement coverage. This recorded information for all the visited classes, methods, lines are stored in a class CoverageCollector. This class is defined to have a HashMap which stores the information for line numbers visited by each test case. We implement the JUnitListener to calculate statement coverage covered by the specific JUnit test in the class JUnitListener. At the end of the JUnit execution, the complete information in this object is written in a file named "CS6367_Phase1\Phase1_Statement_Coverage.txt".

As shown in Figure 1, we ran the command "mvn clean install" in the project folder which created a jar file in the .m2\repository folder with the complete path of "user\.m2\repository\org\example\JavaAgent\1.0-SNAPSHOT\JavaAgent-1.0-SNAPSHOT.jar".

Figure 1

This jar file need to be copied in the root directory of the target project using the command.
$ cp target/JavaAgent-1.0-SNAPSHOT.jar [destination]

To calculate the statement coverage we need to add the configurations as shown in Figure 2 to the build plugins tag in the pom.xml.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <configuration>
      <argLine>-javaagent:./JavaAgent-1.0-SNAPSHOT.jar</argLine>
      <properties>
         <property>
            <name>listener</name>
            <value>CS6367_Phase1.JUnitListener</value>
         </property>
      </properties>
   </configuration>
</plugin>
```

Figure 2

Now we simply run the command "mvn test" from the main project folder having the pom.xml file in it. Maven will automatically download all the required jar dependencies along with the jar for our project and execute the command.

After successful execution of mvn test, the following Figure 3 will appear on the terminal.

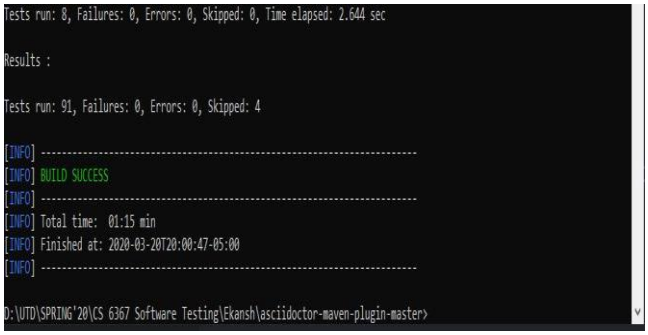

Figure 3

It displays the number of tests run, failures, errors and skipped tests along with the total time taken.

After completion of all the test cases, irrespective of whether they were successful or not, a file "CS6367_Phase1\Phase1_Statement_Coverage.txt" is created in the main project folder depicting the statement coverage information for each test in the format: Test Case Number : 'x' Class Executed: classname:methodname & Line Number: linescovered.

The output of the test run on the asciidoctor project is shown in Figure 4 below.

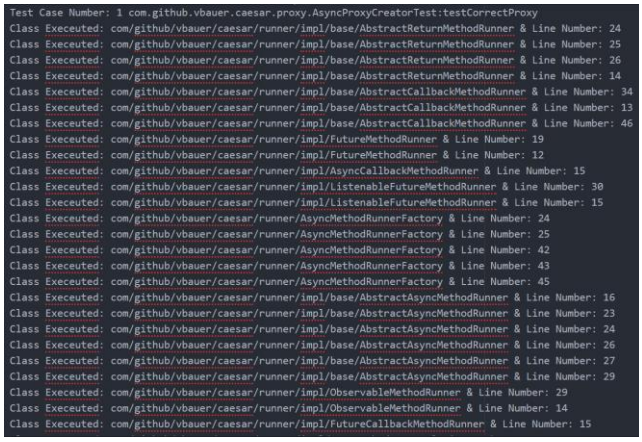

Figure 4

4.2 Report

The results of test run on GitHub repositories downloaded from the web using the list given by professor as a reference, are inserted in the following table:

Project Name	**Test Cases**	**Time Taken**
commons-lang	5912	67.0 s
asciidoctor	91	75 s
brickhouse	45	7.143 s
geo	73	36.679 s
hivemall	63	12.659 s
deft	83	29.357 s
lookup	56	6.242 s
natty	55	12.243 s
protonpack	87	27.378 s
compile-testing	216	20.096 s

Table-1

5. Phase-II

5.1 Concept

An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it. Invariants are widely used in the following areas:

- Theory of optimizing compliers.
- Used as oracle/assertions for testing/verification.

Invariant Patterns

The table below explains the type of invariants that will be covered by the tool.

Single Variables	
Constant Value	x = z
Uninitialized Value	x = uninit
Small Value Set	x ∈ {i,j,k}
Single Numeric Variables	
Range Limits	x >= i, x <= j, etc…
Non-zero	x != 0
Modulus	x = i (mod j)
Non-Modulus	x != i (mod j)

Table-2

5.2 Existing Techniques

There are many tools that can be used to find program invariants.

5.2.1 Daikon

Daikon invariant detector reports likely program invariants. It runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions.
It is an implementation of dynamic detection of likely invariants. It can detect properties in C, C++, Java, etc.

The process is:

- Instrument the program to write data trace files.
- Run the program on a test set.
- Offline invariant engine reads data trace files, checks for a collection of potential invariants.

5.2.2 Stanford Invariant Generator

- StInG is primarily a research tool and is constantly evolving. It implements three techniques for the invariant generation problem. Two of them belong to the more traditional Propagation-based techniques for invariant generation. These are contrasted against our own approach to the problem that we call the Constraint-based technique. The tool has been built only as a platform to test ideas. A new release will contain at least two more algorithms for invariant generation and compilers/translators for many more benchmark examples.

- StInG computes invariants by constraint solving. In effect it reads in the transition system description and compiles them into constraints. Any solution to these constraints is an invariant. StInG has two main components, (1) Compiles linear transition system into Parametric-Linear Constraints, and (2) a heuristic constraint-solver that uses a lazy Constraint-Logic

Programming like technique to solve non-linear constraints of a very special structure.

- Compilation:
  The compilation into constraints is achieved by using a very standard mathematical technique called dualization. A linear invariant for the program looks like this template below:

$$c1\ x + c2\ y + d >= 0$$

the idea is to specify constraints on the unknown c1,c2 and ds so that any solution to the constraints when plugged-in to the template will yield an invariant.

5.3 Implementation

Our first step was to trace method parameter value and we used the following approaches to attain that:

- **visitMethod** method:
  This method redeems the access and name of a method. It is a part of ClassVisitor class. We are overriding it and implementing our own logic In MyClassVisitor class. However, we are not getting the output as expected.

Syntax of **visitMethod ()**:

```
@SuppressWarnings("unchecked")
@Override
public MethodVisitor visitMethod(final int access, final String name, final String desc, final String signature,
    final String[] exceptions) {
```

Figure

- **visitFieldVariable** method:
  In our initial run we tried to override **visitFieldInsn ()** of **MethodVisitor** class and in result we got the indexes of all the fields but could not get the values of those variables in our result.

  Syntax of **visitFieldVariable ():**

```
public static void visitFieldVariable(String className, String methodName, String variableName,
String variableType , String variableValue){
```

Figure

Result which gives the name, and signature of the field variables:

- **visitLocalVariable** method:
  Here, we decided to override the **visitLocalVariable ()** method of the class **MethodVisitor** and called the method as soon as any other method in the code is invoked. This call is to fetch all the local variables of the target folder.

  Syntax of **visitLocalVariable ():**

```
public static void visitLocalVariable(String className, String methodName,
String variableName, String variableType, String variableValue){
```

- **visitCode** method:
  This method is use to fetch all the field and local variables from the classNode object, Since, classNode object contains all the information regarding methods, variables and fields related to that class. But, we were not able to retrieve all the data.
- **isMethodStatic** has value 0 if the method is an instance method, and thus there is an increment to the value. The increment in the value means that the '**this**' argument is present in the method.

  Implementation of **isMethodStatic** mentioned above:

```
boolean isStaticField = (field.access & Opcodes.ACC_STATIC) != 0;
```

6. Experimental Evaluation Subjects and design

Every project should have an experimental evaluation as software should have the possibility to grow with the time and need. Hence, our project will consist of two things:

- Our project will be able to successfully compute the statement coverage by working on the developed collection on the projects chosen from GitHub for testing.

- It also will be able to successfully compute the branch coverage once the functionalities mentioned in Phase-II of the coverage tool are added onto at the end of the aforementioned phase.

REFERENCES

[1] International Journal of Software Engineering and Its Applications Vol.8. No.5 (2014). Examining the Effectiveness of Testing Coverage Tools: An Empirical Study. IJSEIA, pp.139-162 http://dx.doi.org/10.14257/ijseia.2014.8.5.12.

[2] https://en.wikipedia.org/wiki/Code_coverage.

[3] https://asm.ow2.io/.

[4] https://www.tomquest.com/blog/2014/01/intro-java-agent-and-bytecode-manipulation/ .

[5] https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html.

[6] https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html .

[7] https://github.com

[8] https://dzone.com/articles/junit-listener .

[9] https://memorynotfound.com/add-junit-listener-example/.

[10] https://github.com/apache/commons-dbutils.

[11] https://github.com/searls/jasmine-maven-plugin.

[12] https://github.com/FasterXML/jackson-datatype-guava.

[13] https://github.com/rnewson/couchdb-lucene.

[14] https://github.com/craigwblake/redline.

[15] https://github.com/rschildmeijer/deft.

[16] https://github.com/kungfoo/geohash-java.

[17] https://github.com/lemire/JavaFastPFOR.

[18] https://github.com/poetix/protonpack.

[19] https://github.com/google/compile-testing.

[20] https://www.utdallas.edu/~lxz144130/idisc.html

[21] https://plse.cs.washington.edu/daikon/