CS6367 Final Progress Report

Praveen Ramani  
 PXR170005

Anjali Sundareswaran Varier  
 AXV180027

Evan Guo  
 EXG140130

ABSTRACT

With the creation of the computer, processes, which would have taken humans an excessive amount of time to work through, could now be solved within a reasonable time frame. As such, humans could now provide certain instructions to the computer as a framework and through this framework, provide inputs into the system to receive a desired output. While this has greatly enhanced the speed, reliability, and availability of processed data, it is not without its flaws. The computer can only do as much as what the framework requires it to do. If the framework is functionally executable but the algorithm is faulty, the computer will not be able to detect the error, leading to faulty results. As such, there needs to be a method to test and diagnose the framework. It is the goal of this project to develop the diagnostic software that can be applied onto the framework and determine whether the framework is complete, error free, and exhaustive.

1 Basis of technique

In order to check the validity of the framework or code, code coverage must be achieved. In order to do this, the code should be split into independent segments, that is, a block or a single line of code where each variable that is used in that segment is used and/or initialized only once. If a certain part of the code has branching paths (i.e. if, while, and switch statements), it either is placed in its own block or at the very least, the last statement in a segment. All segments should have a path coming from or pointing to another segment. Some segments may have multiple paths coming from it, such as if and else if statements. Other segments may have multiple paths pointing to it, such as a while statement or the segment that follows an if/else statement. All segments are weakly connected to one another, i.e. there exist a path between any two segments provided that the path allows bidirectional travel. This structure of segments and paths is referred to as a flow diagram and from this diagram, we can analysis certain portions of the diagram. This can be done by using coverage techniques such as control flow coverages which checks whether every segment, branch, and path is viable(possible). Other techniques include data-flow coverages, which looks at the variables/data in the flow diagram and determines whether the data is manipulated properly/responsibly.

Additionally, the code should also be tested based on inputs. Code coverage, by itself, merely checks to see what lines of code are covered during a test suite run. Furthermore, code coverage does not say how many times a method may be called during a test case as any duplicate line number is ignored. For this reason, a data trace should be used to determine what values are inserted into the methods and what variables are being accessed inside the method and what values they contain. By having this data trace, we would then be able to develop some invariants about the method, such as the range of values a certain variable would contain, whether a variable is constant, whether it depends on values from the parameters in order to be initialized, etc.

From this analysis, we will then be able to develop a test suite that covers over these criteria and produce an exhaustive traversal of the framework.

2 Implementation

2.1 ASM

Using ASM, we will be able to translate the subject’s project Java files into Java bytecode which we will then be able to change and manipulate without altering the source code. Furthermore, the output of the bytecode is similar to the syntax of the original file, meaning the bytecode is still intelligible enough to be read without an additional agent. ASM is also capable of providing data flow analysis and type-checking algorithms which will be useful in determining whether the subject’s project is complete and error-free. Lastly, ASM has helper classes, that can aid in the verification of the tested code source. In this project we will extend a few of those helper classes to create our own custom visitors. Called “ClassTransformVisitor” and “MethodTransformVisitor,” these two custom classes extend from “ClassVisitor” and “MethodVisitor” respectively, allowing us to override some of their functions which we can modify for our own benefit. These two classes together allow us to track which classes/methods the test cases are traversing on the test project’s java files. Furthermore, we will also be able to track and record what line numbers each test case covers and then send that information to another class within our project to create our code coverage report.

2.2 Java Agent

When using ASM, a significant amount of time is needed for the reading and writing between the bytecode and the file system. To help shorten this time, Java Agent will be used so that ASM’s processes will occur and be compiled in memory, drastically decreasing processing time while leaving the file system unchanged. Furthermore, Java Agent can be combined directly with ASM for a cleaner and faster way to do code and bytecode instrumentation on the fly. For this project, a java file called “Agent” will be used to accomplish this task. Within this java file, we will have to specify the path towards the test project’s test cases as each unique test project has its own unique path towards its own test cases. Failure to do so will prevent the test project from running its own test cases on itself, thereby preventing code coverage. This java file also contains the code that will run our custom ASM transform files, which will be needed to create the modified bytecode and track statement coverage.

2.3 JUnit Listener

In order to record coverage information for each JUnit test case in the test suite over a subject’s project, we will be using a JUnit Listener. A java file called “JUnitExecutionListener” will be created to accomplish this a task. Currently implemented in that java file are 4 methods, each one that shall run depending on the current status of the test project’s test cases. There is a method for when a test suite begins, when a test case starts, when the test case ends, and finally when the test suite finishes. Each method will call upon another custom class called “ControlFlowCoverage” and use its method to store the name of the test class, the coverage information from the test cases as well as create a text document called “stmt-cov” (short for statement coverage) when the test suite completes or ends. These two files/classes are separated to promote security and ease of modification without hampering the entire class.

2.4 Maven

Since we will be using our software validation framework on some sample projects retrieved from GitHub, we first have to able to compile those GitHub projects. Typically to compile those projects, we first need to have access to the same libraries those projects are using, and set the paths accordingly. Maven will be used to solve this issue as it will automatically download those libraries and create the paths for us, allowing us to swiftly build the sample projects. When we download a test project from GitHub, we will use the “mvn test” command as, on first run, it will accomplish the tasks that was stated earlier in this block, then run the test project’s test cases onto itself and check whether they are runnable, error-free. Further use of the command will only run the test cases so long as the test project has not been “cleaned” or modified. We will also use Maven to create our own Maven Project that will contain the code needed to conduct code coverage on the test projects. We will modify in our personal project’s pom xml file the plugins that the project will use. All java files that we create that are related to code coverage will be stored in the src/main/java folder. When all java files have been created and written and the pom xml has been formatted, we will then use the “mvn build package” command to create a jar file of our project that will then be referred to and used in the test project, after specifying how in the test project’s pom xml and using “mvn test.”

**2.5 Daikon**

Daikon is used for determining the potential invariants. Invariants are the properties that hold at a certain point, often seen in assert statements, documentations and formal specifications. Daikon runs a program, observes the values that the program computes and reports what are true over the observed executions. We are mainly focusing on the program internal states in the second phase. We plan to extend the current coverage detection tool to include invariant detection by implementing Daikon. It involves tracing every variable value at all method entries, obtaining single variable and single numeric variable invariants etc., The tool will be tested on apache’s commons-lang project and the results will be analyzed.

3 Study Plan

Phase 1:

Requirements gathering: Feb 21 – Feb 27

Gathering test files: Feb 28- March 3

Creating the coverage collection tool: March 4 – March 10

Analyzing results and report creation: March 11 – March 13

Phase 2:

Augmenting program to trace internal states: March 14 – April 5

Analyze and improve the program: April 7 – April 10

Analyzing results and report creation: April 11 – April 16

**4** **GitHub Projects used for testing the tool**

1. <https://github.com/nodebox/nodebox>
2. https://github.com/zeroturnaround/zt-exec
3. <https://github.com/maven-nar/nar-maven-plugin>
4. <https://github.com/jfree/jfreechart-fse>
5. <https://github.com/smartrics/RestFixture>
6. <https://github.com/damianszczepanik/cucumber-reporting>
7. <https://github.com/NLPchina/ansj_seg>
8. <https://github.com/EngineHub/CommandHelper>
9. <https://github.com/apache/commons-codec>
10. <https://github.com/TooTallNate/Java-WebSocket>

**5** **Phase I Results Obtained**

1. <https://github.com/nodebox/nodebox> :

Tests Covered: 304 tests

Size of the statement coverage text file (stmt-cov.txt): 2049 K

Total run time: 46.465 s

1. https://github.com/zeroturnaround/zt-exec:

Tests Covered: 96 tests

Size of the statement coverage text file (stmt-cov.txt): 677 K

Total run time: 39.708 s

1. <https://github.com/maven-nar/nar-maven-plugin>:

Tests Covered: 234 tests

Size of the statement coverage text file (stmt-cov.txt): 729 K

Total run time: 01:11 min

1. <https://github.com/jfree/jfreechart-fse>:

Tests Covered: 2192 tests

Size of the statement coverage text file (stmt-cov.txt): 22289 K

Total run time: 37.415 s

1. <https://github.com/smartrics/RestFixture>:

Tests Covered: 290 tests

Size of the statement coverage text file (stmt-cov.txt): 1537 K

Total run time: 15.673 s

1. <https://github.com/damianszczepanik/cucumber-reporting>:

Tests Covered: 338 tests

Size of the statement coverage text file (stmt-cov.txt): 5183 K

Total run time: 15.261s

1. <https://github.com/NLPchina/ansj_seg>:

Tests Covered: 59 tests

Size of the statement coverage text file (stmt-cov.txt): 997 K

Total run time: 15.261s

1. <https://github.com/EngineHub/CommandHelper>:

Tests Covered: 705 tests

Size of the statement coverage text file (stmt-cov.txt): 31187 K

Total run time: 1.20 min

1. <https://github.com/apache/commons-codec>:

Tests Covered: 1085 tests

Size of the statement coverage text file (stmt-cov.txt): 2760 K

Total run time: 3.99 s

1. <https://github.com/TooTallNate/Java-WebSocket>:

Tests Covered: 286 tests

Size of the statement coverage text file (stmt-cov.txt): 30712 K

Total run time: 1.18 min

6 Evaluation

The project will be tested on at least 10 projects (with >1000 lines of code) with JUnit tests (with >50 tests) from GitHub to collect the coverage for its JUnit test suite. The evaluation will be based on both efficiency and effectiveness. Efficiency shall be defined as the time it takes for our project to cover and traverse the subject projects. Effectiveness will be defined as the completeness of the coverage as well as correctness based upon the results of the test suite. To assist with the evaluation of our project, we will look at and use commonly used control flow coverage techniques such as statement, branch, path, method, stack, class, etc. Additionally, we will also use data flow coverages include All-Defs, All-Uses, All-DU Paths, All P-Uses, All C-Uses, etc.

In phase 1, we will be focusing our efforts on control flow and data flow coverage.

In phase 2, we will augment the coverage collection tool by tracing information on the internal state of the subject’s program.

7 Execution

7.1 Phase I

For phase 1 of the project, we needed to write code that would collect the line coverage of a sample project by using its test cases. To begin, we used the classes given to us from the professor and reference websites to construct our base.

The Agent class and JUnitExecutionListener class were taken from their respective reference website and modified to fulfill our needs. For example, the Agent class was lightly modified, only specifying the name of our personal class visitor and telling it which file pathways it needed to access in order to get to the test suite/ test cases of our sample projects. From JUnitExecutionListener, we used the methods already present in the file as based upon how those methods were named, we assumed they directly correlated to what happens during the execution of a test suite. To clarify, we used the class as an observer/mechanism that directly determined exactly what our project should do given a certain phase of the test suite. For example, whenever a test suite begins, we start initializing our code coverage platform. Whenever a test case begins, we prepare the platform to start observing the test case and store information of what lines the test case covers. Lastly, whenever the test suite finishes, we instructed our project to output the location, methods, and lines that a certain test suite covers onto a text file in order to determine the extent of the test suite’s line coverage.

Two other class files, ClassTransformVisitor and MethodTransformVisitor, were taken from a previous homework assignment given by the professor. Based upon our knowledge of how ASM operated, we believed these two classes were enough in order to use byte code manipulation to retrieve the line numbers. Little modification went into ClassTransformVisitor as getting line numbers was based primarily in MethodTransformVisitor. The only change that was made was getting the name of currently visited class and sending it over to MethodTransformVisitor. While working in MethodTransformVisitor, we experienced an issue of trying to understand how to get the line number and store them as a variable because based on the homework where it originated from, it was only getting outputted to screen. From examination of the code and research from the internet, we learned that the “visitLdcInsn” and “visitMethodInsn” were the two primary attributes of MethodVisitor class that could fulfill the task of retrieving the line number and inputting it into a variable. After creating a test function in another class, we performed some trial and error on this function, which eventually lead to successful transmission of line numbers to another class via an ASM based method.

Knowledgeable of how to send data via byte code manipulation, we continued to add functions onto the test class, which then became our most important class for tracing and printing line number coverage, which shall henceforth be called “ControlFlowCoverage.” Within this class, we defined the variables needed to properly identify and store certain line numbers. All methods within this class were controlled or influenced by JUnitExecutionListener and performed certain task such as relating line numbers to certain classes and methods as well as relating those classes and methods to certain test cases that called on them. Once the test suite on the sample project ended, the print method would be called, which will create, format, and add to a statement coverage text file and be of similar structure to that of a reference text file provided by the professor.

During this phase of the project, the main issue that plagued our group was understanding how to use Maven, as no one has had any previous experience with it before. This generated a whole host of issues such as downloading maven from the appropriate website, setting up appropriate environment paths, integrating Maven Project into Eclipse IDE, referencing certain jar files with our Maven Project, formatting our project’s pom xml, compiling a Maven Project, packaging the project , its contents and referenced jar files into an external jar file, and finally formatting the sample project’s pom xml to use our own project’s generated jar file and Agent.

In conclusion, most of the time spent during this phase of the project was used creating the Maven Project and running it than that of writing the Coverage Collection code that would be used inside of the Maven Project.

7.2 Phase II

Phase 2 of the project involved creating a data trace of the sample project as well as determining its invariants. This involved using the Daikon techniques to determine certain invariants based on the values that are used or stored within the methods. Based upon our initial understanding of the requirements, we believed that Daikon had to be incorporated into our project to retrieve the invariants. Later clarification revealed that this was not the case and that creating our own code that reflected the Daikon technique was acceptable.

For the first leg of this phase, our group focused primarily on data tracing as we could later use the data trace to determine invariants. This was done by using the ASM’s tree node classes. Within “ClassTransformVisitor” we used a variable type “ClassNode” to grab an instance of the class that the program was operating in. Once retrieved, it would then be able to decompose the ClassNode into a list of MethodNode (each containing one of the class’ methods), for which each MethodNode could further be decomposed into a list of LocalVariableNodes (each containing a local variable of a certain method). Prior to the decomposition of the MethodNodes into a list of its LocalVariableNodes, the program first must check to see if the MethodNode matches the method that the class is currently visiting. Once it is, then decomposition can take place and the list of LocalVariableNodes is sent to MethodTransformVisitor for further processing. The “visitCode” of MethodTransformVisitor is used for extracting data from LocalVariableNode because, according to the ASM documentation, it runs at the beginning of a method invocation, meaning this code should be executed prior to any other method that MethodVisitor runs normally. Inside this overridden method, it first checks whether the currently visited method is static or not as this determines which stack index it should begin at. By default, the LocalVariableNode does not explicitly state what data type it is and what value it contains. To retrieve the type, the Type variable is used along with the local variable descriptor to determine concretely what type it is supposed to be. To retrieve the value, “visitVarInsn” is used along with a stack index mentioned earlier to retrieve the local variable instruction on the stack which will then be called upon by the appropriate “visitMethodInsn,” which is based upon the Type, to convert retrieved stack frame into the appropriate value. Once this is completed, the current class’ name the variable is located in, the current method’s name the variable is located it, the method descriptor, the variable’s type, the variable’s name, and the variable’s value are sent to another class, called Data Tracing, to be stored. It should be noted that this process only retrieves the parameters of the method and its values as the group was unable to develop a method to retrieve them.

To retrieve the fields of the class, FieldNodes were used to accomplish this task. Using the same ClassNode from earlier, it was possible to decompose a ClassNode into a list of FieldNodes, which would also be sent to “visitCode” in MethodTransformVisitor for processing. Inside “visitCode”, a separate, user created method was called which takes in each FieldNode from the list and extract the specific FieldNode’s type and that of its value in a process similar to that performed on LocalVariableNode mentioned earlier, however with the additional step of checking if the field is static and using the appropriate Opcode in “visitMethodInsn”. Following this, the class name the fields originated from, method name, method descriptor, field type, field name, and field value were also sent to Data Tracing class to be stored.

A class called Data Tracing was created to perform a similar role to that of our ControlFlowCoverage class. Within this class, the class name and method name received from MethodTransformVisitor will be used as a key to retrieve all the methods and its associated variables. The method descriptor would also be used as a secondary key since overloaded methods do exist within some sample projects. Finally, the variable type and name would serve as the final key to gain access to a HashSet that contained the values that were every assigned to that specific variable. A secondary HashMap was created to keep track of the number of times a certain variable has been accessed. Once the test suite completes its course, the Data Tracing class will then create a text file listing all field variables of a class as well as the variables of a given method along with their values.

For the process listed above, starting from the beginning of section 7.2, it should be noted that this was a simplified, streamlined process of events taken to produce a functioning data trace and does not mention the numerous issues encountered during the development. Such issues will be listed in the next paragraph.

For the issues encountered during this phase of the project, we shall list the largest problem that has plagued the group the most. To explain, the most daunting issue was the constant presence of some error which would occur during the running of a Maven Test on a sample project. The errors of “StackOverflowError” and “NoSuchMethodError” were the most ubiquitous and the hardest to solve as our understanding of these errors at the time was that it was happening on the test suite side. The presence of these errors typically caused the test case to fail and prevent the capture of both line number and variables which results in a smaller, incomplete data trace and statement coverage text files. Our group eventually managed to resolved these errors and learned these errors happened because sometimes the variable type of the local or field variables were not always primitive data types but rather Objects, such as classes or arrays, that stored a reference to a data which was initialized in the sample projects. Therefore, whenever a reference object variable was encountered during a run, our project would try to store the reference object as a String, which were not always successful as on regular occasion the reference object was a class that was defined within the sample project. As such, in order to resolve the issue, all Objects were given a hash code for its value as a substitute to its actual value.

7.2.1 INVARIANT DETECTION

Once all the local as well as field variables and their values are found, we analyzed them to find their invariants. This was done in the DataTracing.java file. For each variable in the method, we iterate over the set of all values it had during the tests and compare them to check if they fit any of the invariant patterns.

For example, if the variable always has only one particular value for all the tests, then we consider that variable to be a ‘constant invariant’.

If the variable is an integer and its values are within a certain range, then we consider the variable to be a range invariant.

If the variable contains no value for all the tests its involved in, then we consider the variable to be an uninitialized invariant and they are marked in the ‘invariants.txt’ file as ‘UnInit Invariants’.

We check for constant, range and uninit invariants for integer types and check constant and uninit for all other types.

Some issues we faced in this section was that we were unable to iterate through the list directly and therefore we had to convert it into a string using .toString() and then parsed the string to get the variable name, type, and the list of values. We split the string using ‘=[‘ as the regex and not ‘=’ as regex because certain values of string type contained ‘=’ in them and so using ‘=[‘ solved the problem. This split the string into two, giving the variable and type in one and the list of values in the other.

We split the first part of the string using “ “ as the regex and we obtain the type and the variable name. For the second part, we first remove the tailing ‘]’ and then split it with ‘, ‘ as the regex and this gives the list of individual values.

If the type is an integer, then we convert the values into integer from string using Integer.parseInt(). We then compare the values with one another to find the invariant pattern.

For other types, we keep the values as strings and then compare them with one another to find the invariant pattern.

7.2.2 CONFIDENCE MEASURE

We have created a confidence metric to measure the confidence of our claims on the invariant variables. We have considered the number of values in the list of values of a variable, which is also the number of times the method (and hence the variables) have been invoked, as the confidence metric.

According to statistical heuristics, a sample size of 30 is considered to be good. Therefore, we consider the claims we made about variables with,

* more than 30 values in the values list, to be of high confidence.
* 11 to 30 values in the values list, to be of medium confidence.
* Less than 10 values in the values list, to be of low confidence.

One issue we faced in this section was deciding on the confidence metric. We decided on the number of values in the list as the metric because it is similar to sample size i.e if more values follow an invariant pattern then higher the confidence.

7.3 Phase II RESULTS OBTAINED

1. https://github.com/apache/commons-dbutils :

Tests Covered: 304 tests

Size of the statement coverage text file (stmt-cov.txt): 967 K

Size of the data trace text file (datatrace.txt): 642 K

Size of the invariants text file (invariants.txt): 421 K

Total run time: 41.672 s

**Sample Output:**

**Datatrace.txt:**

org/apache/commons/dbutils/DbUtils.loadDriver:-->  
  
StringdriverClassName=[]  
ClassLoader classLoader=[2f7ad930]

**Invariants.txt**:

org/apache/commons/dbutils/DbUtils.loadDriver:-->  
  
String driverClassName ---> UnInit Invariant --No. of values analyzed(Confidence)= 1 --Low Confidence

ClassLoader classLoader = 2f7ad930 ---> Constant Invariant --No. of values analyzed(Confidence)= 1 --Low Confidence

**Explanation**:

* Since the driverClassName variable has no values, it is an uninitialized invariant
* Since the classLoader variable has only one value and nothing else to compare to it is considered to be constant invariant.
* Since both have only one entry, these are low confidence claims.
* org/apache/commons/dbutils/DbUtils.loadDriver
  + org/apache/commons/dbutils/DbUtils – class name
  + loadDriver – method

7.4 PROJECT LINK

We had some difficulties using the github and as a result, we were unable to co-ordinate the project code in github and we had to resort to co-ordinating by other means. Therefore, the individual contribution history is unavailable.

In order to make the code visible to you, we uploaded the completed project to the github account of one of our members and the link is provided below:

<https://github.com/PravGitHub/CS-6367>

REFERENCES

[1] http://asm.ow2.org/

[2] https://github.com/

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

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

[5] http://www.tomsquest.com/blog/2014/01/intro-java-agent-and-bytecode-manipulation/

[6] http://memorynotfound.com/add-junit-listener-example/

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

[8] <http://https://elearning.utdallas.edu/bbcswebdav/pid-3083096-dt-content-rid-80926732_1/courses/2202-merged-CS6367001-SE6367001/course-project.pdf>

[9] <https://plse.cs.washington.edu/daikon/>