Nihar Gupte
1001556441
Summary and Discussion

The course project for CSE 4321 – Software Testing and Maintenance, taught at The University of Texas at Arlington by Dr. Jeff Lei during the Fall of 2020 provided an opportunity to practice the basic concepts, principles and techniques covered during the course of the semester. The students were provided with a java program of moderate size – "printtookens.java" which had about 500 lines of code with 15 seeded faults. The purpose the program was to implement a string tokenizer, and thereby classify the input tokens into variety of types such as string constant, numeric constant, character constant, identifier, keyword, comment, special characters and so on. The input could be taken in either of the two ways – either using command line console input or using input read via a file stored in local directory. The corresponding output would be type of each token, one per line.

Before I began the project, I thoroughly read through the project description and made sure that I have understood the requirements well. I did so by asking any questions that I had to the professor / my fellow classmates, and then downloading and setting up my local environment needed for this project. The primary steps involved in the development of this project were:

1. Creation of CFGs

The first step in order to begin the project was to identify the basic blocks for the methods which have at least one branching decision in them. Also, the code in catch blocks was skipped for the purpose of this project. For user-defined functions such as those utilized in the main(), I identified them under their own separate basic block. For the lines of code, which had multiple statements on a single line, the first statement was labeled as the 'a' part, and the second statement as the 'b' part of that line. After identifying all such basic blocks, I proceeded to make Control Flow Graph (CFG) Diagrams for the methods specified above. I used the online tool of "app.diagrams.net" which was recommended to me by the professor. It was easy to use as I could conveniently duplicate nodes and arrows, and then save the diagram. The branching condition – true or false was written on the arrow label. Thus, the CFGs were created and imported as .png pictures into the file – "CFG.pdf", along with their respective code snippet screenshot and the basic block tables. There were 15 such methods, and hence 15 graphs drawn (including the main method)

2. Achieve Edge Coverage

The second step was to select the test paths from the CFGs drawn above so that the paths achieve edge coverage. For the purpose of this project, edge coverage was required to be achieved only for six methods – get_token, is_token_end, is_keyword, is_num_constant, is_str_constant, and is_identifier. Thus method-level method or unit-testing was performed by achieving edge coverage for these six methods. Similarly, program-wide edge coverage was also achieved for for the CFG of the main method, in order to do a program-level testing by covering all the edges in all the CFGs. Other methods were skipped. Edge coverage was achieved by going through each one of the six methods and the main method individually, and then making sure that all the

edges appeared in at least one of the test cases. The test paths are labeled in the document "Test Cases.pdf", underneath their respective functions.

3. Generate Test Cases

After selecting all test paths to achieve edge coverage for the CFGs, test cases were generated. Each test case included the test data i.e. the input data, and the expected output for that test path. Each test case began with the Start node, i.e. the 0^{th} node and ends with the End node, as per its corresponding CFG detailed in "CFG.pdf." Additional clarification about the format and reporting used for the main method is given in the document "Test Cases.pdf." The main method achieves program wide edge coverage and hence the chosen format was used in order to abstract complex test paths into simple test paths. Also, for this document, the test cases were chosen in such a way that a selected test path had only one corresponding test data input, however in practical JUnit testing, I used multiple similar inputs of the same type to achieve a higher coverage percentage. For example, in the test cases with input data of "\n" is replaceable by input data of "\t" or "\r" in practical JUnit testing.

4. JUnit testing

The next step was to use JUnit to implement and execute the generated test cases and find out the faults. However, while performing the earlier steps such as CFG generation and especially Test Case generation, I noticed a variety of faults in the program, simply by static analysis. Hence, I fixed them first before proceeding to JUnit testing. I wrote test cases to test the non-main methods and the main method separately.

For the selected six non-main methods, I performed their JUnit testing in the file "NonMainMethodsTest.java". Six test methods, corresponding to the six non-main methods were used in the testing file. Each test method made sure to achieve edge coverage and also test multiple similar inputs of the same type of data in order to achieve higher percentage of coverage. For 5 of the methods, I was able to achieve 100% coverage, while for get_token, I achieved about 96% coverage.

For the main method, I performed the JUnit testing in the file "MainMethodTest.java" using 51 test methods. This was performed by a combination of input from the console as well as file reading. Each method in the testing java file corresponds to the same serial number of the test case as written in "Test Cases.pdf". Additionally, multiple similar inputs of the same type are tested using function names such as test4a, test4b, test4c, and so on. I was able to achieve a total coverage percentage of 95.6% for the entire printtokens.java along with many individual methods reporting a coverage percentage of 100%. The testing for the method main_test2a() was done using the file "emptyFile.txt" and for method main_test2b() was done using the file "test.txt". (These files are attached along with other files of the project).

5. Branch Coverage Reports

Since Eclipse comes preinstalled with JaCoCo – a library for Java Code Coverage, I used the IDE of Eclipse to generate Branch Coverage Reports based on the JUnit test. This was very helpful as I could visually see which branches from the program were covered by the tests and which were not. I could use this vital information in order to increase the coverage percentage by covering the uncovered branches/lines of code.

“UnitTestingCoverageReport” corresponds to the branch coverage report for the unit tests of the six individual methods. “ProgramLevelTestingCoverageReport” corresponds to the branch coverage report for the program-level tests of the main method. These reports are zipped under “CodeCoverageReports.zip”.

Based on the results from JUnit tests and static analysis as done before while generating CFGs and test cases, I was able to find out about 22 faults in the given java program. These faults are simple index errors, typographical errors, cases of missing code block, and so on. The faults, along with their location, and fixes is given in the document “Faults Detected and Corrections.pdf”.

I used the IDE of Eclipse mainly because it comes with support for JaCoCo as well as JUnit testing. System.exit(0) was terminating the JVM, and so to fix this issue, I used the concept of pseudo return statement in java.

In the JUnit Source code, along with the “MainMethodTest.java” and “NonMainMethodsTest.java”, I have also included the corrected version of the java program – “Printtokens.java”. The initial version with seeded faults is also provided for reference as “original-Printtokens.java”. A ReadMe file is also provided to help execute the JUnit code.

In conclusion, the list of submitted files for this project include:

1. CFG.pdf
2. Test Cases.pdf
3. Source Code (directory)
    a. MainMethodTest.java
    b. NonMainMethodsTest.java
    c. Printtokens.java
    d. original-Printtokens.java
    e. emptyFile.txt
    f. test.txt
4. Code Coverage Reports.zip
    a. UnitTestingCoverageReport.zip
    b. ProgramLevelTestingCoverageReport.zip
5. Faults Detected and Corrections.pdf
6. Summary and Discussion.pdf (this file)
7. ReadMe.txt

This project helped me in applying the theoretical concepts that I learned via the course lectures and homework assignments to a real-world scenario. I now have a better understanding of how software is tested in the industry. Learning JUnit and testing procedures will certainly be beneficiary to me – both, on my resume and during technical interviews as I graduate and step into the job market.