COSC 442
Software Quality Assurance & Testing
Spring 2018

Project #5 – Test Generation for Finite State Machines using Chow's Method

"What kind of programmer is so divorced from reality that he/she thinks they'll get complex software right the first time?"
- James Alan Gardner

Directions: This project has 3 tasks that will guide you through testing software systems based on finite state machines using Chow's method (i.e., the W method). **Please read through all instructions prior to starting**.

Goals: The intention of this project is to become more familiar with testing software systems based on finite state machines using Chow's method (i.e., the W method) and tool support to generate test cases using the W method algorithm from the book. In addition, this project should continue to familiarize yourself with several software development tools (Eclipse, Git, Javadocs and GitHub). Secondly, this project is again intended to put the process of testing into the context of the software engineering lifecycle by developing test cases from a requirements/design artifact (e.g., a finite state machine). In doing so, you should further develop skills in test case design, program specification, unit testing as well as practical experience in programming and software development tools within a larger software engineering process.

Points: 20

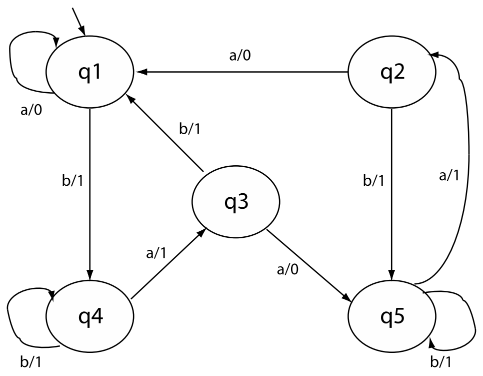
Deadline: Wednesday, April 25, 2018 with all source code and project artifacts checked into your GitHub repository.

Language/Environment Requirements: Java within the Eclipse IDE using JUnit 4.12 and the W method tool provided can be used for this project.

Task 1 – Chow's Method Tool – Book Example. In lecture, we manually went through the W-method algorithm to generate the test cases for a specified finite state machine (FSM). In doing so, we needed to calculate the characterization set (W), the transition cover set (P), as well as determine the input and output alphabet. This process, done manually, is error prone and not realistic in a real-world setting. Thankfully, industrial-strength tools (e.g., Rational Rose, IBM Rhapsody, etc.) automate test generation algorithms, like the W-method algorithm, for FSMs. For this project, we will use a small tool, provided by the author, that implements the W-method algorithm for a specified FSM.

Download and import the WMethod.zip Eclipse project file from Blackboard (Projects / Project 5). This project has two directories: a source directory with the implementation of the W-method algorithm and a directory with a number of example FSMs specified.

To become familiar with using this tool, we will first run an example FSM using the FSM from the book:



Recall that the FSM notation used here is of the form x/y where x denotes the input to the FSM and y the corresponding output. For this tool, a specified FSM needs to be encoded and saved as a plain text file. The test generation tool reads the FSM from a specified file. In the file specifying an FSM, data for each edge occupies one line, separated by a newline character. Each line contains, in order, the label of the source state, the label of the destination state, and the edge label. Edge labels are treated internally as strings of characters. While the state labels must be positive integers. The edge label must be of the form x/y where x denotes the input to the FSM and y the corresponding output.

So, for the book's example, the FSM specification file would be as follows:

```
1 1 a/0
1 4 b/1
2 5 b/1
2 1 a/0
3 1 b/1
3 5 a/0
4 4 b/1
4 3 a/1
5 5 b/1
5 2 a/1
```

This encoded FSM specification file is provided in the root directory, as BookExampleFSM.txt, of the WMethod tool project. To run the WMethod tool for this example FSM, compile and run the `WMethod.java` file and provide the FSM file name when prompted. After doing so, the WMethod tool will output the following results:

```
Enter filename: BookExampleFSM.txt
FSM input from: BookExampleFSM.txt
States: 5
Edges 10
Input alphabet:
a
b

Output alphabet:
0
1

From     Input/Output   To
1        a/0            1
1        b/1            4
```

```
2       a/0          1
2       b/1          5
3       a/0          5
3       b/1          1
4       b/1          4
4       a/1          3
5       b/1          5
5       a/1          2

Transition cover set (P). 11 entries.
Empty a b ba baa baaa baaaa baaab baab bab bb

W Set. 4 entries.
a aa aaa baaa

Number of Test Cases :29
Test cases: [a, aa, aaa, aaaa, abaaa, ba, baa, baaa, baaaa, baaaaa, baaaaaa, baaaaaaa,
baaaabaaa, baaaba, baaabaa, baaabaaa, baaabbaaa, baaba, baabaa, baabaaa, baabbaaa, baba,
babaa, babaaa, babbaaa, bba, bbaa, bbaaa, bbbaaa]
```

NEW! This output goes throught the W-method algorithm to produce a test set adequate for testing the software system based on this FSM. In addition, the tool has a method in the `Utilities` class named `runFSM()` with following method signature

```
public static void runFSM(State [] FSM, int stateID, String input, String separator)
```

where `FSM` is an array containing the states of the FSM to be executed, `stateID` is the ID of the state from the execution is to begin, `input` is the string of input symbols to be applied to the FSM, and separator is a string that separates the elements in the input string. For example, given an FSM, the following command executes it against the input "a a b a b":

```
Utilities.runFSM(FSM, 1, "a a b a b", " ");
```

and would yield the following output:

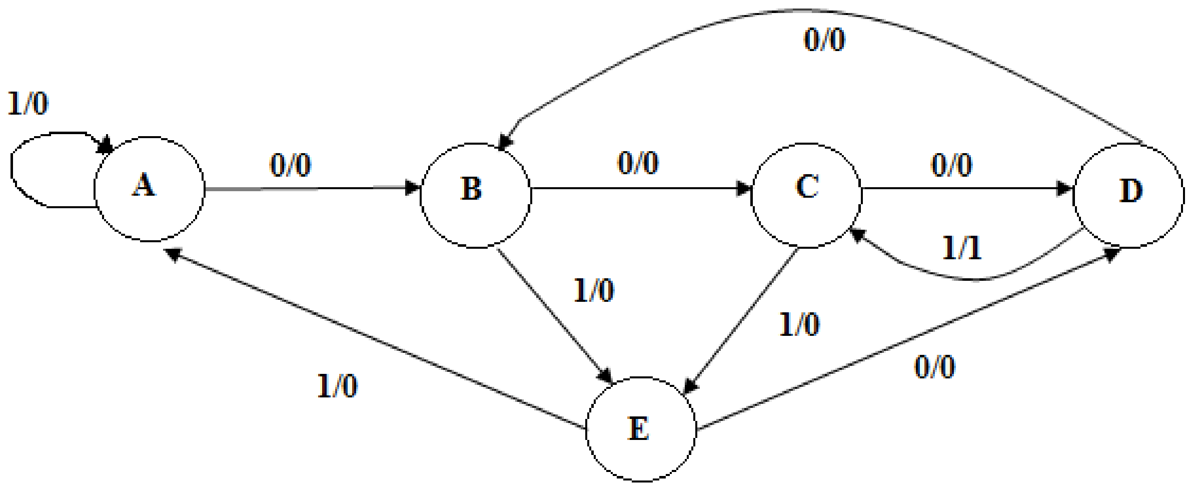
```
FSM execution begins. Input: a a b a b Initial state: 1
Current state: 1
 Input: a Next state: 1 Output: 0
Current state: 1
 Input: a Next state: 1 Output: 0
Current state: 1
 Input: b Next state: 4 Output: 1
Current state: 4
 Input: a Next state: 3 Output: 1
Current state: 3
 Input: b Next state: 1 Output: 1

FSM execution completed. Final state: 1
Output pattern:00111
```

One feature that this current tool lacks is the ability to run the generated test cases against the specified FSM, despite having the functionality to do both. For this task, complete the Eclipse TODO (in `WMethod.java`'s main method to add this functionality. That is, you should take the test cases stored in the `tests` Vector and feed them into the `Utilities.runFSM()` method. Once you have completed this, rerun the WMethod file so that it generates the test cases and then runs each against the machine. Create a directory, named Task1, and include a text file with the console output.

Once you have completed this task, you should remove the TODO comment, save the output file in the Task1 directory and commit this project to your repository (with appropriate commit comments!).

Task 2 – VHDL Bit Stream Detector. Consider the following FSM that detects the overlapping sequence "0X01" in a bit stream:



where A is the initial state and the "1" output indicates that the FSM detected the sequence 0X01 in a bit stream. Encode this FSM in as a specification input file for the WMethod tool and generate the test set and output of running the test set cases against the FSM using the code your developed in Task 1. Note, this tool wants states labeled starting with integers starting with 1 – so don't label the initial state for this FSM as A. When you have completed this, create a directory, named Task2, and include a text file with the console output and your input file.

Once you have completed this task, you should commit this project to your repository (with appropriate commit comments!).

Task 3 – Bond, James Bond. The final task of this project will have you develop your own FSM based on requirements, experience test-first development and hacking/engineering this tool to nearly-automatically generate JUnit test cases based on a FSM.

The use of regular expressions is common, powerful programming skill for any developer and is one that is ripe with mistakes. Regular expressions can, conveniently, be modeled as a FSM. Consider a string, consisting of only digits (i.e., 0-9) and the right and left parentheses of which you want to know whether the sequence "007" is found in the string between a left and right parentheses. That is, *(*007*)* where * represents zero or more digits or parentheses. The only allowed input symbols are: 0-9, (and). The output symbols are "no" anytime that the desired sequence (i.e., *(*007*)*) has not been discovered and "yes" when the desired sequence has been discovered.

First, develop/draw the FSM based on these requirements and include it in the project under a directory named Task3. It is fine to draw and take a picture of your FSM as long as the input/output symbols on each edge are clear.

Once you have an FSM drawn, encode this FSM in as a specification input file for the WMethod tool and generate the test set and output of running the test set cases against the FSM using the code your developed in Task 1. If done correctly, the output should indicate more than 200 test cases for this FSM. When you have completed this, include a text file with the console output and your input file in the Task3 directory.

It would be *extremely* tedious to develop the JUnit test methods for all 200+ test cases generated by the W method tool. However, because we have access to the W method source code, we can somewhat easily generate the JUnit test methods programmatically, similar to what you did in Task 1. Develop the code, in the WMethod's main method to output the JUnit methods needed to test all the generated test cases for a `bondRegex` method that takes in a string input and returns a Boolean depending on if the desired sequence was found in the input string or not. To do so, consider the following hints:

- A simple for loop iterating through all the elements in the `tests` vector will give you the test values
- A `System.out.println("“"public void testCase" + i + "(){");` will print out the beginning of the generated JUnit method for a test case
- To determine whether to print assertTrue or assertFalse, you will likely need to modify the Utilities.runFSM method to get access to the outputPattern variable and search (i.e., `contains(“yes”))` and suppress the `println` statements in the `Utilities.runFSM` method.

When you have completed this, create a new Eclipse project, named Project5 and an appropriate package name where you create a new class, named `JamesBond.java`. Before creating the implementation of the `bondRegex` method in the `JamesBond.java` method, you should create the unit test code file, named `JamesBondTest.java` where you include the JUnit test code you generated in a legal, compilable JUnit file. This will be the unit test code that will test a method, with the following signature:

```
public Boolean bondRegex(String input)
```

When you have the test code developed, you should develop a bondRegex implementation that passes all unit tests. Your implementation must utilize Java's regular expression functionality. For help with Java regular expressions, see:

- http://www.vogella.com/tutorials/JavaRegularExpressions/article.html
- https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html
- http://www.tutorialspoint.com/java/java_regular_expressions.htm
- http://www.regular-expressions.info/java.html

Once you have completed this task, you should commit both the Project5 project and the updated WMethod project to your repository (with appropriate commit comments!).

Grading: This project will be graded out of 20 (2 for Task 1, 3 for Task 2 and 15 for Task 3) possible points.

Submission: You must submit your reports, source code and test cases to your GitHub repository. Failure to submit the project following these guidelines may result in your project not being graded.

Note: There are at least 3 Easter eggs (this isn't one of them) in this project. If you find one, let me know via direct message on Slack. Happy hunting! ☺