Testing the Spatiotemporal Epidemiological Modeler (STEM)

Noah Drake, Chloe Harris, Tristan Lawler, Ryan Ratliffe

Dr. Jim Bowring – CSCI 362.01, Fall 2020

Introduction

This project is the result of a semester-long project for a Software Engineering class. The project utilizes the H/FOSS (Humanitarian / Free and Open Source Software) project called The Spatiotemporal Epidemiological Modeler (STEM). From the STEM wiki, "The Spatiotemporal Epidemiological Modeler (STEM) tool is designed to help scientists and public health officials create and use spatial and temporal models of emerging infectious diseases. These models can aid in understanding and potentially preventing the spread of such diseases." In short, STEM provides a framework so that developers and researchers can model disease infections and run simulations ranging from a local to a global scale. This can be used to determine what policies and strategies may help slow or prevent the spread of various diseases under different conditions.

This project works with STEM in order to test segments of the STEM source code and to generate and return a test report.

Background

All work was performed using either Ubuntu Linux or a Debian-based Linux Subsystem, both using a Bash shell. Languages used include Bash, Java, HTML, and CSS.

In order to build a testing framework for the STEM project, it was necessary to download and configure STEM in a developer environment within the Eclipse IDE. All team members were able to get STEM running within Eclipse, but various issues arose when using Maven to build the project independently of Eclipse. These primarily involved dependencies that could not be resolved, but no two team members got exactly the same error and we were unable to fully resolve this.

For those that could not get Maven to build STEM properly, they were able to retrieve a working copy of the code from those that could. The group then found the required .jar files for dependencies so that everyone could work with portions of the code without having to rebuild the entire project. Because of this, it is now possible to run our testing suite without having to interface directly with the original STEM project. In order to test certain classes and methods, more .jar files would need to be added so that the proper dependencies are in place—we only included a subset.

In order to utilize the testing framework we generated, the code only needs to be cloned from our GitHub. Instructions are included in the read me for adding more test cases and for fault injection. Our script includes a build script, the testing script, and the reporting script, which are all utilized in our runAllTests.sh script.

Our process for this project, which will be discussed in more detail in the following sections, included developing and documenting a testing framework, working with a specified framework structure, injecting faults into the code, and having our code generate an HTML test report that provides a compiled summary of each test run.

Testing Framework

Our test automation framework utilizes a Data-Driven Framework model. A driver was designed for each method that we test, and our testing script reads inputs and other relevant information (including which driver to use) from a test file. This allows each test to be run with multiple data sets just by modifying an appropriate test file and including it in the correct directory. Because test data is not hard coded into the scripts, it is easier to include new test cases for methods after the driver has been implemented.

We worked within a pre-defined framework directory structure, shown in Figure 1, that was provided to us. The root directory, called TestAutomation, held subdirectories for project dependencies, script files, test case files, documents, etc. The objective was for the tests to be run and a report generated when the command "./scripts/runAllTests.sh" was run from the Test Automation top level directory. The script would then access the folder of test case specification files.

```
/TestAutomation
        /project
                /src
                /bin
                / …
        /scripts
                runAllTests.(some scripting extension)
                ... other helper scripts
        /testCases
                testCase1.txt (or xml or json)
                testCase2.txt (or xml or json)
                …
        /testCasesExecutables
                testCase1(may be folder or file)
                ...
        /temp (for output from running tests … be sure to clean at start of runAllTests)
                testCase1results (might be folder or file)
                ...
        /oracles
                testCase1Oracle (might be folder or file)
                ...
        /docs
                README.txt
        /reports
                testReport.(txt or html)
```

Figure 1. Framework Directory Structure used

The test case specification template provided included the following information, with each piece of information on its own line in a text file: * test number/ID, requirement being tested, component being tested, method being tested, test input(s) including command-line argument(s), and expected outcome(s)*. This information, along with the expected outcome from a separate oracles directory, was compiled to be presented in the test report. The format of our HTML test report is demonstrated in Figure 2.

Test suite run at Tue Dec 1 10:17:08 EST 2020

Test Case 001 - Passed	
Driver	org.eclipse.stem.test.driver.neldermeadalgorithm.NelderMeadTestDriver
Component	org.eclipse.stem.analysis.automaticexperiment.NelderMeadAlgorithm
Method	execute()
Requirement	Function minimization
Test Input(s)	0, 0.0, 9999999.0, 1, 0.0, 9999999.0, 1.8, 1.2, 0.5, 0.5, SampleFunction2
Expected Results	2, 1, -4
Computed Results	[2, 1, -4]

Figure 2. HTML Test Report Sample

Testing Process

Our testing process was designed around 5 deliverables that were required to be completed throughout the semester. These were:

1. Checkout or clone project from its repository and build it. Run existing tests and collect results.
2. Produce a detailed test plan. Specify at least 5 of 25 test cases.
3. Design and build an automated testing framework that will implement the test plan.
4. Complete the design and implementation of the testing framework. Create at least 25 total test cases.
5. Design and inject 5 faults into the code that will cause at least 5 tests to fail. Ideally not all tests will fail.

The most time was spent on deliverable 1, simply getting the STEM project source code up and running. While we thought this would be easy given the extensive documentation provided on the STEM wiki, we learned that even good documentation can fall short when it comes to getting code to work on various systems. Version differences and dependencies caused the biggest roadblocks, especially working with two different versions of java at the same time. Secondarily, building the project outside of an eclipse development environment had dependency challenges for every member of the group, but there was not any documentation in this regard since it was out of scope for most development.

The next part of the process was also time consuming—after getting the code up and running, we had to parse through it to find classes and methods for testing. STEM is a large project, and it was not always clear where certain classes and methods were defined. STEM utilizes a lot of inheritance and lots of interfaces, which did make it less accessible when first working with the code.

The design of our automated testing framework was not complete until the final implementation fault injection because we kept hitting unexpected obstacles along the way. We started with three main scripts, a build/cleaner script to ensure old files were cleared each time the test suite was run and to make sure the most recent version of the project was built, the test script itself, and finally a test reporter script that would create and display our report. As the project advanced, we were able to combine these three scripts into one. This script would clean old files, build the project, run the tests, then compile the test report and display it in a web browser. The format of our test cases is shown below in Figure 3.

```
001
Function minimization
org.eclipse.stem.analysis.automaticexperiment.NelderMeadAlgorithm
execute()
org.eclipse.stem.test.driver.neldermeadalgorithm.NelderMeadTestDriver
0, 0.0, 9999999.0, 1, 0.0, 9999999.0, 1.8, 1.2, 0.5, 0.5, SampleFunction2
2, 1, -4
```

Figure 3. Test File format used to run our test suite.

* The content in each line is described in the previous column above Figure 2

Injecting Faults

All tests passed when working with the original STEM code, as was expected. To make sure that our test would properly report whether a test had passed or failed, we injected faults into STEM.

To accomplish this, we had to change the way we integrated the STEM source code into our project. We did this by deleting every .class file that we were testing in our dependencies and including the .java files for these classes in our src folder instead. This means the script now compiles the STEM source code being tested along with our drivers and supporting classes.

The faults injected were as follows:

org.eclipse.stem.graphgenerators.impl.PajekNetGraphGeneratorImplOld

- line 1022: Remove "- x"
- (will cause all test cases to fail)

org.eclipse.stem.analysis.automaticexperiment.NelderMeadAlgorithm

- Line 44: Change 0.5 to 0.1
- (will cause test case 004 to fail)

org.eclipse.stem.analysis.impl.ReferenceScenarioDataMapImpl

- Line 911: Change Math.abs(2.0*(d1-d2)/(d1+d2)) to Math.abs(2.0*(d1+d2)/(d1-d2))
- (will cause test cases 007 and 008 to fail)

org.eclipse.stem.core.math.BinomialDistributionUtil

- Line 71: Change p = p/100 to p = p*100
- (will cause test cases 012 and 014 to fail)

org.eclipse.stem.core.model.impl.STEMTimeImpl

- Line 104: Change
  final long newTime = newTime().getTime() + timeIncrement; to
  final long newTime = newTime().getTime() - timeIncrement;
- (will cause test cases 016, 018, 019, and 020 to fail)

References and Acknowledgements

Thanks to Dr. Jim Bowring who taught our CSCI 362 Software Engineering class and to the Eclipse Foundation and developers of STEM for their H/FOSS work.

https://wiki.eclipse.org/STEM
https://github.com/csci-362-01-2020/Team-4

Poster template (design Newfield) from
https://www.posterpresentations.com/free-poster-templates.html