SyncTest: An Approach To Unit Testing Multicore Software

UNDERGRADUATE HONOURS THESIS
FACULTY OF SCIENCE (COMPUTING SCIENCE)
UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Alexander D. Marshall

SUPERVISOR:
Jeremy S. Bradbury

April 20, 2016

Abstract

As computer hardware advances and becomes more complicated, computer software will also become more complicated to make use of this new hardware. One way software developers are making use of better hardware is by writing concurrent programs that utilize multi-core processors. In a concurrent program, multiple threads work in parallel to achieve a goal whereas a sequential program only uses a single thread to achieve that goal. The advantage of concurrent programs is that they are more efficient than sequential programs and can complete more complicated tasks. The downside, however, is that they are prone to many bugs that make testing and debugging far more complicated. There are many tools for unit testing software but they are not equipped to deal with concurrent programs. There are also several tools for debugging concurrent programs but they are not meant to be used as testing tools.

In an attempt to bridge the gap between these two areas of interest, this thesis has researched and developed a command line tool and an Eclipse plugin for unit testing multi-threaded Java programs by combining a unit testing tool and a concurrency debugging tool. While the command line tool and plugin are fully functional, they act as more of a proof of concept and are better suited for use on small scale programs.

Contents

List of Figures

List of Tables

1 Introduction

One of the most important facets of computer science and software engineering is the idea of testing. A piece of code or software is useless if it doesn't function as intended and the best way to ensure proper functionality is to test extensively on different levels of detail. When dealing with concurrency however, typical testing methods are not sufficient and more specific tools must be used to uncover bugs. The purpose of this thesis is to discuss these two areas of importance and how they can work together to test and optimize concurrent code.

The goal of this thesis was to create a tool for unit testing concurrent Java code, by building on existing tools including Eclipse, JUnit, and SyncDebugger. This tool was created in two main phases: creating a Linux bash script that would act as a proof of concept and then turning this script into a plugin for the Eclipse IDE. The bash script has the desired functionality in the sense that it runs Java unit tests while detecting and terminating deadlocked threads. What it lacks however, is a user interface for accepting inputs and providing useful outputs. The plugin functions in the same way the script does and additionally has an interface for the user to configure how the plugin works and to display the test results in a way that is useful to the user.

This thesis will provide a background for understanding the reasoning behind the creation of the plugin, the development process of the bash script and plugin, as well as the future work that can be done on the plugin.

2 Background and Motivation

In this section I will provide a brief background on unit testing in Java followed by concurrency in Java and the types of bugs that can occur when dealing with concurrency.

2.1 Unit Testing

Unit testing is a comprehensive form of software testing which tests code on the class or function level of an object-oriented programming language. Unit tests are typically written by the same developers that write the code being tested because they have the greatest understanding of how it works. The purpose of unit testing is to ensure that each individual piece, or unit, of code works individually before integrating them to form a larger system. There are dozens of unit testing frameworks for hundreds of programming languages but this thesis will focus on JUnit, the primary unit testing framework for Java programs. JUnit is an excellent tool that is use widely by Java developers but it is particularly weak in one area: concurrency. Concurrency bugs appear intermittently and so trying to detect them through a sequential testing framework simply does not work well.

2.2 Concurrency

In computer science and software engineering, concurrent programs are ones that do not run in a sequential order but instead have multiple processes, or threads, acting in parallel to achieve a goal more efficiently. Since these programs do not run sequentially, the order that the units of code execute does not matter and the result will remain determinate. The main issue with concurrent systems is that they do not always achieve the same result and can result in a variety of bugs

```
synchronized void transfer(Account ac, double mn) {
    if(amount - mn < 0) {
        System.out.println("ERROR - Insufficient funds for transfer"    );
        System.exit(1);
    }

   if(ac.num < num) {
       synchronized(ac) {
           synchronized(this) {
                amount -= mn;
                ac.amount += mn;
           }
       }
   } else {
        synchronized(this) {
           synchronized(ac) {
               amount -= mn;
               ac.amount += mn;
           }
       }
   }
}
```

Figure 1: Deadlock Example

An example of Java code that will result in a deadlock. This method is for a bank account program and its purpose is to transfer money from one bank account to another. To make a transfer it must have locks on both accounts, but a deadlock can occur when two accounts are each waiting for the lock on the other.

including deadlocks and race conditions. Some of the main concurrency bugs include:

- Deadlock (Figure 1):

  "...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle. For example, this occurs when a thread holds a lock that another thread desires and vice-versa" [LSW07].

- Livelock:

  "...similar to deadlock in that the program does not make progress. However, in a deadlocked computation there is no possible execution sequence which succeeds, whereas in a livelocked computation there are successful computations, but there are also one or more execution

sequences in which no thread makes progress” [LSW07].

- Interference (data race, race condition):

  ”...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous.” [LSW07].

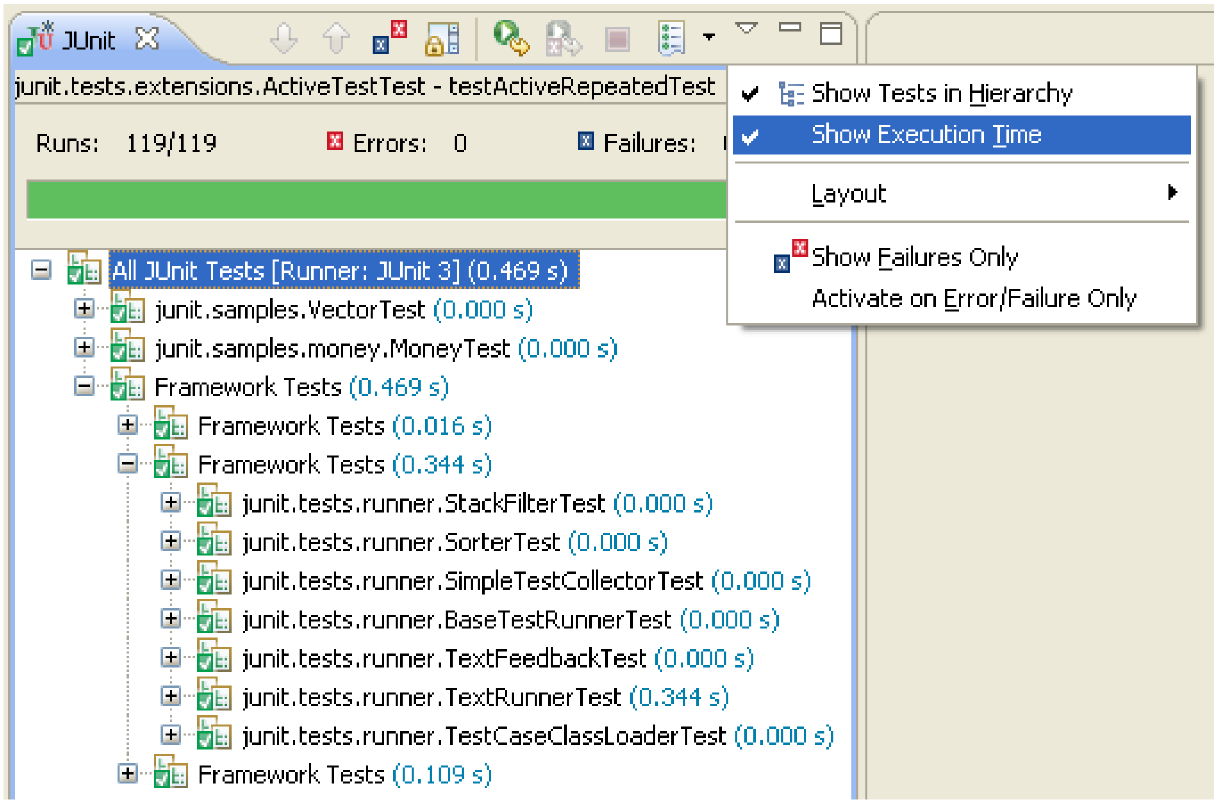
Another important area of concurrency is the ’schedules’ that threads run on, which are determined by how the threads interleave. Say there are two threads each with three tasks to complete in any order. The different schedules for this example are simply the different sequences in which these six tasks are completed. In a program with perfect concurrency, the schedules won’t matter and the result will remain determinate. In the real world, however, this isn’t always the case. A large part of optimizing concurrency is determining which schedules cause bugs and either fixing that bug or preventing that specific schedule from occurring.

There are various tools for optimizing concurrent systems that implement a variety of methods, the most relevant one to this thesis being instrumentation, which is the process of inserting ’noise’ into the code which causes the program to behave differently and possibly cause a bug. I will discuss this more in Section 3 where some of these tools will be introduced.

2.3 Motivation

The problem that now arises is that unit testing frameworks are not equipped to detect bugs in concurrent code and the tools for optimizing concurrent code are not meant to be integrated with these testing frameworks. Additionally, these concurrency tools test on the system level and not the unit level, so they are not as thorough as we might like them to be This leads to the primary motivation for this thesis:

To create tools that will both run Java unit tests and detect concurrency bugs. These tools will take the form of a bash script and an Eclipse plugin called SyncTest, which will instrument and unit test concurrent code as well as display the results in useful format.



junit.tests.extensions.ActiveTestTest - testActiveRepeatedTest

Runs: 119/119 Errors: 0 Failures:

```
All JUnit Tests [Runner: JUnit 3] (0.469 s)
  junit.samples.VectorTest (0.000 s)
  junit.samples.money.MoneyTest (0.000 s)
  Framework Tests (0.469 s)
    Framework Tests (0.016 s)
    Framework Tests (0.344 s)
      junit.tests.runner.StackFilterTest (0.000 s)
      junit.tests.runner.SorterTest (0.000 s)
      junit.tests.runner.SimpleTestCollectorTest (0.000 s)
      junit.tests.runner.BaseTestRunnerTest (0.000 s)
      junit.tests.runner.TextFeedbackTest (0.000 s)
      junit.tests.runner.TextRunnerTest (0.344 s)
      junit.tests.runner.TestCaseClassLoaderTest (0.000 s)
    Framework Tests (0.109 s)
```

Figure 2: Eclipse JUnit Plugin

A screenshot of the JUnit plugin for the Eclipse IDE. This was the basis for the design of the SyncTest plugin.

3 Related Tools

In this section I will introduce JUnit, the most widely used unit unit testing framework for the Java programming languages. I will also introduce some tools for detecting concurrency related bugs.

3.1 JUnit

JUnit has been important in the development of test-driven development and is part of a family of unit testing frameworks typically refered to as xUnit. In JUnit, tests are marked with the '@Test' tag and a user can also specify '@Before' and '@After' tags which indicate that certain sections of code should be run before or after each test as well as '@BeforeClass' and '@AfterClass' tags which indicate that these sections of code should be run before or after all of the tests. JUnit has many interesting features but I find the third-party extensions to be the most interesting.

JUnit has an abstract class called Runner, which is used to run the tests that are specified by the user. Testers can create subclasses of Runner to suit more specific needs such as the Hierarchical Context Runner which supports context hierarchies in JUnit. Two custom runners that are highly relevant to this thesis are the ConcurrentTestRunner which runs tests in parallel and the IntermittentTestRunner which runs tests repeatedly to expose intermittent failures [1].

JUnit also has plugins for several IDEs including IntelliJ, NetBeans, and Eclipse (Figure 2). The JUnit Eclipse plugin was a major inspiration for the design of SyncTest because I had decided that if SyncTest would be similar in function to JUnit than it should also be similar in appearance. This would make the plugin feel more familiar to Eclipse and JUnit users.

[1]https://github.com/junit-team/junit4/wiki/Custom-runners

3.2 ConTest

ConTest is a tool developed by IBM, designed to expose and eliminate concurrency related bugs in larger scale, distributed software. The goal of ConTest is to alter the schedules of a program to create a scenario in which bugs like deadlocks and race conditions are more likely to occur. This allows testers to find bugs earlier in production when there are much cheaper to fix. ConTest supports traditional test coverage models and can be used to measure test coverage. An interesting feature of ConTest is the replay feature which attempts to reproduce bugs be recreating the scenario in which the bug originally occurred.

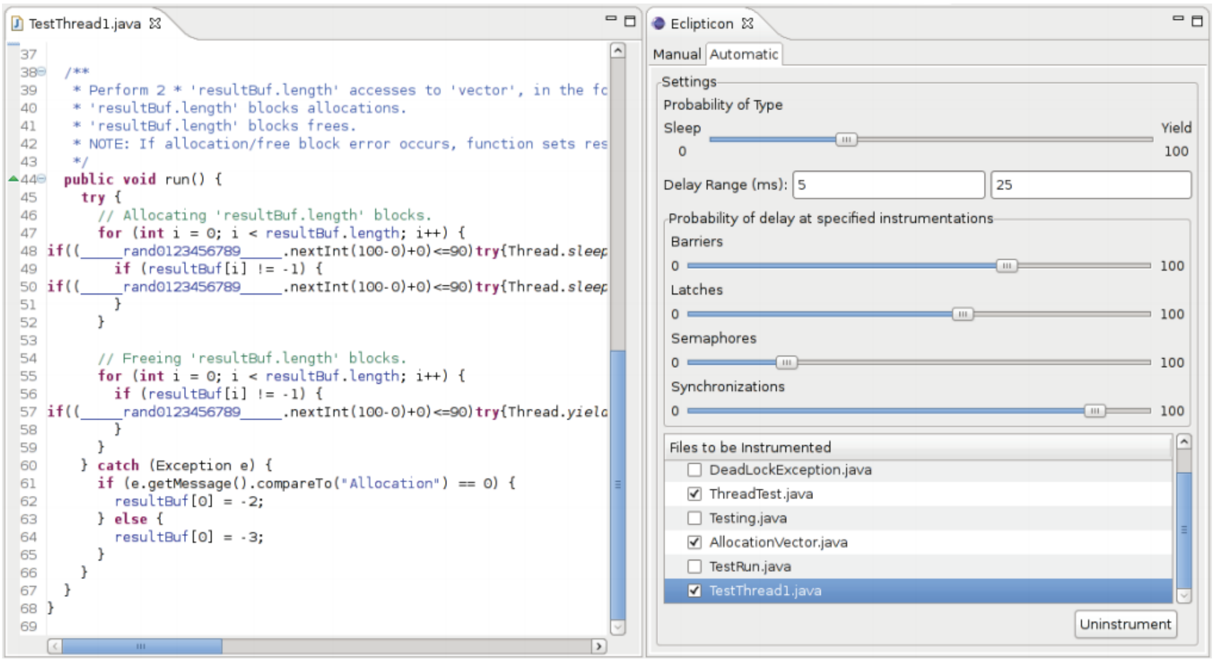
On healing concurrent programs:

"In testing try to make them more likely to fail;
in the field we may want to make them less likely to fail." [U08]

3.3 Eclipticon

Eclipticon is an Eclipse plugin for optimizing concurrency that was developed here at UOIT in the Software Quality Research Lab (Figure 3). It can automatically insert noise into code before running JUnit tests, or a user can manually configure the noising process, giving a finer level of control of the concurrency. This allows the user to more easily localize concurrency bugs and speed up the error correction process.

The noise that Eclipticon inserts takes the form of 'sleep' and 'yield' commands. In Java, the sleep command will cause the current thread to become not runnable for a certain amount of time. During this time, the CPU can run another task or simply idle if there are no more tasks. The yield



```
    /**
     * Perform 2 * 'resultBuf.length' accesses to 'vector', in the fc
     * 'resultBuf.length' blocks allocations.
     * 'resultBuf.length' blocks frees.
     * NOTE: If allocation/free block error occurs, function sets res
     */
    public void run() {
      try {
        // Allocating 'resultBuf.length' blocks.
        for (int i = 0; i < resultBuf.length; i++) {
if((_____rand0123456789_____.nextInt(100-0)+0)<=90)try{Thread.sleep
          if (resultBuf[i] != -1) {
if((_____rand0123456789_____.nextInt(100-0)+0)<=90)try{Thread.sleep
          }
        }

        // Freeing 'resultBuf.length' blocks.
        for (int i = 0; i < resultBuf.length; i++) {
          if (resultBuf[i] != -1) {
if((_____rand0123456789_____.nextInt(100-0)+0)<=90)try{Thread.yield
          }
        }
      } catch (Exception e) {
        if (e.getMessage().compareTo("Allocation") == 0) {
          resultBuf[0] = -2;
        } else {
          resultBuf[0] = -3;
        }
      }
    }
}
```

Settings

Probability of Type

Sleep 0 — Yield 100

Delay Range (ms): 5 25

Probability of delay at specified instrumentations

Barriers 0 — 100

Latches 0 — 100

Semaphores 0 — 100

Synchronizations 0 — 100

Files to be Instrumented

- [ ] DeadLockException.java
- [✓] ThreadTest.java
- [ ] Testing.java
- [✓] AllocationVector.java
- [ ] TestRun.java
- [✓] TestThread1.java

Figure 3: Eclipticon Plugin

A screenshot of the Eclipticon plugin running in Eclipse. The tabbed view for SyncTest was based on the tabbed view in Eclipticon.

command works in a similar way but instead of stopping execution immediately, it acts as a hint to the scheduler that the current thread is willing to yield its current use of a processor[2]. Since thread schedules are very important when discussing concurrency, changing these schedules one of the main ways that bugs are detected. Eclipticon gives users coarse and fine control over the placement of thread delays: coarse control allows users to target entire files whereas fine control allows users to target specific areas of files.

[2]java.lang.Thread documentation

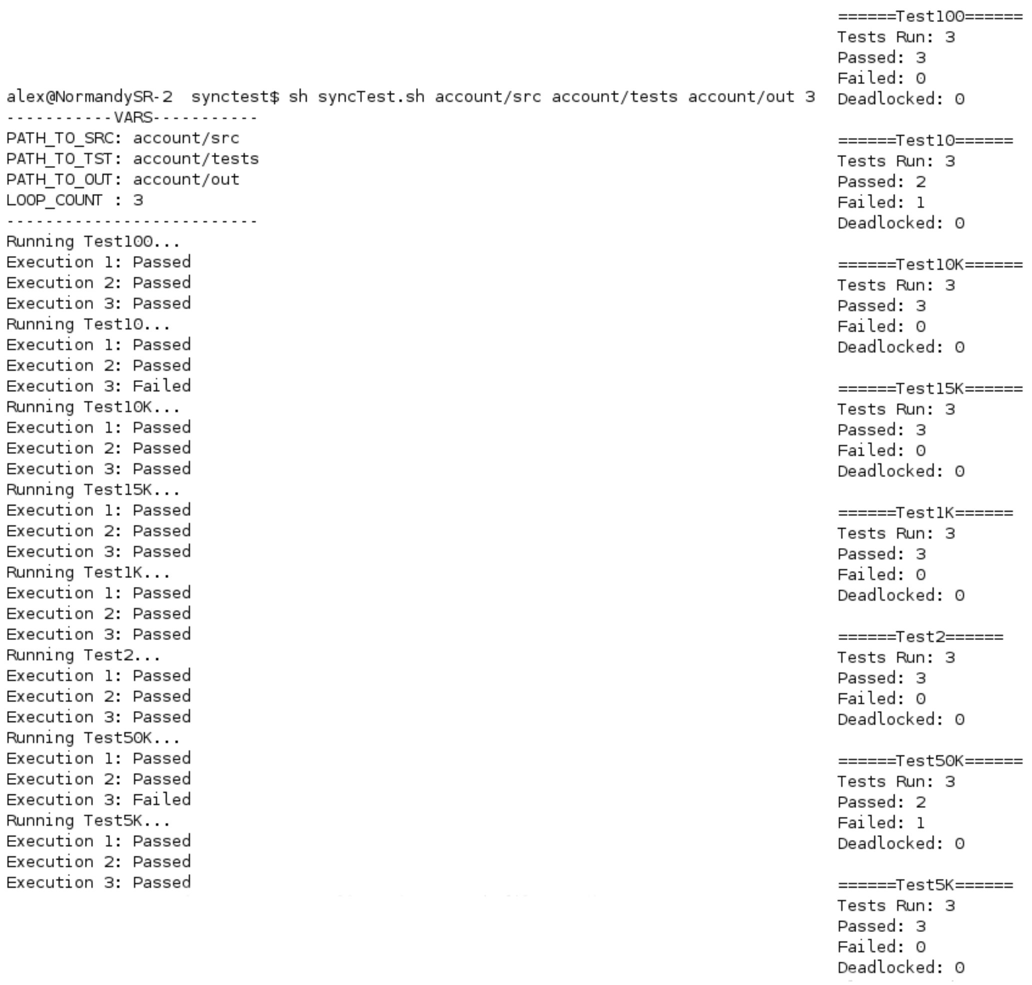
3.4 SyncDebugger

SyncDebugger is an automatic debugging tool for multithreaded Java programs which runs TXL and bash scripts in combination with Java grammar files. The TXL scripts are used to instrument code in a similar way to Eclipticon, except it uses only sleep statements and the amount of time each thread sleeps is random. SyncDebugger is not unlike a binary search: it instruments and runs the entire program, finds which half of the program the bug occurred in, and then runs the program again with only that half noised, continuing until it finds the line of code that is causing the bug. The Java grammar files are used by the TXL scripts to identify key sections of Java code for instrumentation.

4 Bash Prototype

The first step I had taken towards creating SyncTest was the creation of a Linux bash script. The very first version of the script could only run JUnit tests for a simple bank account program in which a series of 2 or more bank accounts would transfer money from one account to the next. The next step was modifying the script so that it could run several JUnit tests multiple times and this was accomplished by using a for-loop to iterate through a directory of test files and a second for-loop to run each test many times. A separate script was used to detect deadlocks by printing the status of all the Java threads for the currently running test and then checking the output for the keyword ”deadlock”. If a deadlock was found, the test would be killed and the original script would move on. This script would be started just before a test was run, and then killed after the test completed.

As each test completes, its output is saved to a file which was later parsed by a Java class that I created. This parser would check all of the outputs from the tests and sum up the results for the user. The final addition to the script was removing all the hard-coded paths and variables, so that the script could take inputs and be used for any Java project, assuming that project was set up in a way that was compatible with the script.



```
alex@NormandySR-2  synctest$ sh syncTest.sh account/src account/tests account/out 3
-----------VARS-----------
PATH_TO_SRC: account/src
PATH_TO_TST: account/tests
PATH_TO_OUT: account/out
LOOP_COUNT : 3
-------------------------
Running Test100...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
Running Test10...
Execution 1: Passed
Execution 2: Passed
Execution 3: Failed
Running Test10K...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
Running Test15K...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
Running Test1K...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
Running Test2...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
Running Test50K...
Execution 1: Passed
Execution 2: Passed
Execution 3: Failed
Running Test5K...
Execution 1: Passed
Execution 2: Passed
Execution 3: Passed
```

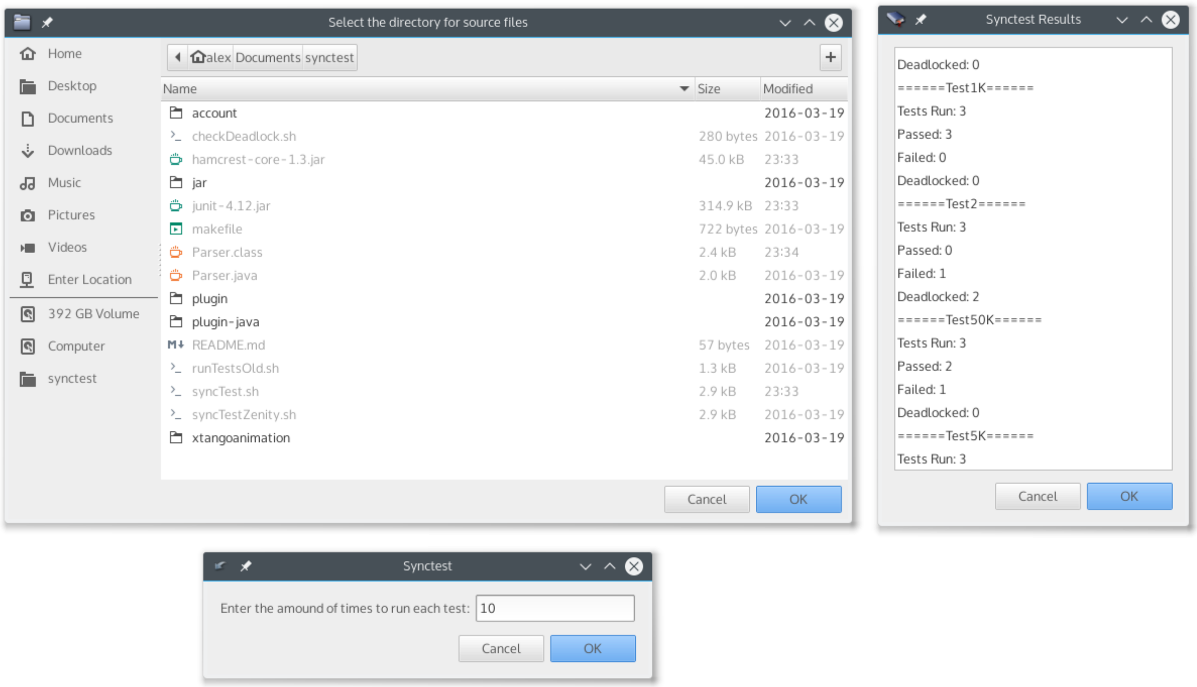
```
======Test100======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0

======Test10======
Tests Run: 3
Passed: 2
Failed: 1
Deadlocked: 0

======Test10K======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0

======Test15K======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0

======Test1K======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0

======Test2======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0

======Test50K======
Tests Run: 3
Passed: 2
Failed: 1
Deadlocked: 0

======Test5K======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0
```

Figure 4: Bash Script Example Inputs and Output

syncTest.sh takes four input arguments: the path to project source files, the path to project test files, the path to a directory for outputs, and the number of times to run each test. The output for syncTest.sh prints the results of tests as they come in as well as a summary of the results after all the tests have completed

4.1 Command Line Interface

The primary version of the script takes all its inputs through the command line and also prints all the results back to the command line (Figure 4). This meets the functionality of the proposed tool, but lacks an interface for visualizing outputs.



Name	Size	Modified
account		2016-03-19
checkDeadlock.sh	280 bytes	2016-03-19
hamcrest-core-1.3.jar	45.0 kB	23:33
jar		2016-03-19
junit-4.12.jar	314.9 kB	23:33
makefile	722 bytes	2016-03-19
Parser.class	2.4 kB	23:34
Parser.java	2.0 kB	2016-03-19
plugin		2016-03-19
plugin-java		2016-03-19
README.md	57 bytes	2016-03-19
runTestsOld.sh	1.3 kB	2016-03-19
syncTest.sh	2.9 kB	23:33
syncTestZenity.sh	2.9 kB	2016-03-19
xtangoanimation		2016-03-19

```
Deadlocked: 0
======Test1K======
Tests Run: 3
Passed: 3
Failed: 0
Deadlocked: 0
======Test2======
Tests Run: 3
Passed: 0
Failed: 1
Deadlocked: 2
======Test50K======
Tests Run: 3
Passed: 2
Failed: 1
Deadlocked: 0
======Test5K======
Tests Run: 3
```

Enter the amound of times to run each test: 10

Figure 5: Example Zenity Dialogue Boxes

The largest window in this screenshot is an example dialogue for selecting a directory. You can see that files are greyed-out so that only folders can be selected. Directly below it is the dialogue for entering the amount of times each test should be run. Finally on the right side is the dialogue which shows the output of the script.

4.2 Zenity Interface

I also experimented with a library called Zenity which allows users to create simple dialogue boxes for command line shell scripts [3]. I used Zenity to create dialogue boxes for accepting the inputs and displaying the outputs. Additionally, I attempted to implement a progress bar but it didn't work in the end. To get the progress bar to function properly, I had to modify the script to provide inputs for the progress bar. The problem that arose was that the progress bar would consume all of the outputs from the script and as a result the script wouldn't work properly. Some example Zenity dialogue boxes are pictured in Figure 5.

[3]https://help.gnome.org/users/zenity/stable/

4.3 Case Study

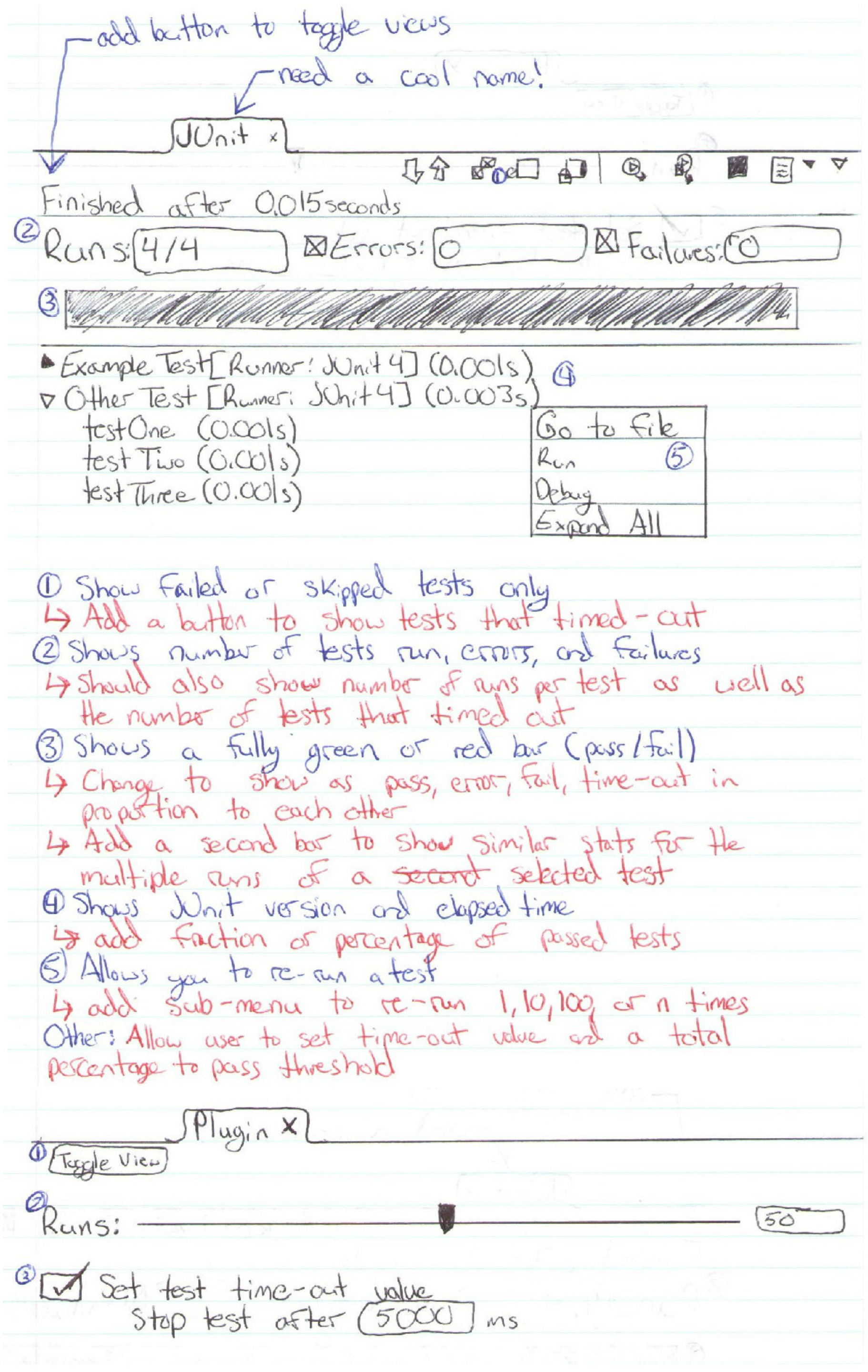
To demonstrate how this script works, we will use the bank account program mentioned in the introduction to this section. This program was chosen because it will result in a deadlock when two accounts are waiting for the locks on each other's accounts (Figure 1). As pictured in Figure 4, we can see that we pass three directories to the script: the directory containing source code, the directory containing tests, and a directory for the output files. The last input is the number of times to run each test. The left column of Figure 4 shows the output of the script as the tests are completed and the right column shows what is printed after all the tests have completed. Similarly, Figure 5 shows the same scenario but with the Zenity dialogue boxes.

5 Eclipse Plugin

In this section I will introduce the Eclipse platform and why it was chosen for this thesis. I will also detail the stages of development of the SyncTest Eclipse plugin starting from early sketches of the interface, then the visual and functional prototypes, and concluding with the major changes that led to the plugin's current state as well as a case study with the bank account program.

5.1 Eclipse Platform

The Eclipse IDE was chosen to be the platform for this thesis for several reasons. First of all, Eclipse is written in Java and we had intended to create a tool for multicore Java code so it made sense to work in Eclipse. Another reason was that Eclipse already had a JUnit plugin which could be used as a basis for the design of the SyncTest plugin. This way, users who are already familiar with the JUnit plugin shouldn't have trouble using SyncTest. Eclipse also has plenty of support for plugin development, so I had no trouble finding documentation and help when creating the plugin. Lastly, I had access to Eclipticon's source code so I examined it in an effort to learn more about how Eclipse plugins work. Eclipticon was also considered as an option for instrumenting code since it is also an Eclipse plugin.



① Show failed or skipped tests only
↳ Add a button to show tests that timed-out
② Shows number of tests run, errors, and failures
↳ Should also show number of runs per test as well as the number of tests that timed out
③ Shows a fully green or red bar (pass / fail)
↳ Change to show as pass, error, fail, time-out in proportion to each other
↳ Add a second bar to show similar stats for the multiple runs of a ~~second~~ selected test
④ Shows JUnit version and elapsed time
↳ add fraction or percentage of passed tests
⑤ Allows you to re-run a test
↳ add sub-menu to re-run 1, 10, 100, or n times
Other: Allow user to set time-out value and a total percentage to pass threshold

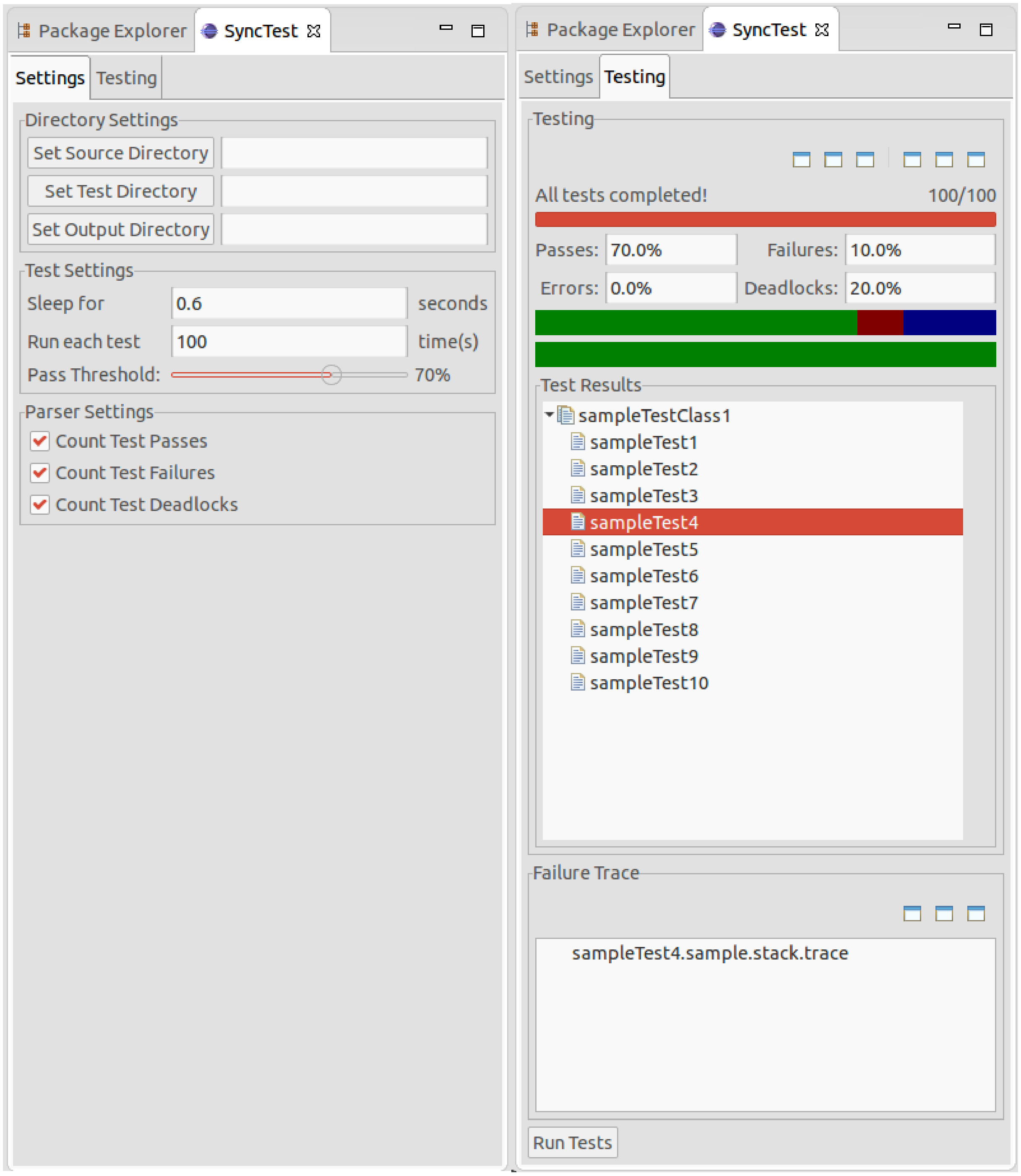
Figure 6: JUnit/SyncTest Sketches

My notes on the JUnit plugin for Eclipse. I ended up going with a text box to enter the number of runs for each test and the test time-out value became the frequency for checking for deadlocks. When I created the visual prototype of the plugin, I based it off of these sketches.

5.2 Interface Sketches

The very first thing I did in developing this plugin, even before the creation of the bash script, was draw up some sketches of what I think the plugin interface should look like (Figure 6). I drew these sketches with no knowledge of how the plugin would work or how to create plugins in Eclipse and so I took the JUnit plugin as a basis and made some notes as to what I thought would be useful improvements.

My main issue with the JUnit plugin is with the green bar (Figure 2). What the bar indicates is that if all of your tests pass the bar is green, but if even one fails the bar becomes red. Since the SyncTest plugin involved running many tests many times, I proposed that the bar would be compromised of several colours, one for each possible outcome of a test, and that they would fill the bar proportionally. I also proposed that SyncTest would have two bars, one for overall results and one that would display results for the executions of individual tests.



Directory Settings

Test Settings

Sleep for 0.6 seconds

Run each test 100 time(s)

Pass Threshold: 70%

Parser Settings

- [✓] Count Test Passes
- [✓] Count Test Failures
- [✓] Count Test Deadlocks

Testing

All tests completed! 100/100

Passes: 70.0% Failures: 10.0%

Errors: 0.0% Deadlocks: 20.0%

Test Results

- sampleTestClass1
  - sampleTest1
  - sampleTest2
  - sampleTest3
  - sampleTest4
  - sampleTest5
  - sampleTest6
  - sampleTest7
  - sampleTest8
  - sampleTest9
  - sampleTest10

Failure Trace

```
sampleTest4.sample.stack.trace
```

Figure 7: SyncTest Visual Prototype

The first version of the SyncTest plugin running in Ubuntu GNOME. It has since undergone many visual and functional changes.

5.3 Visual Prototype

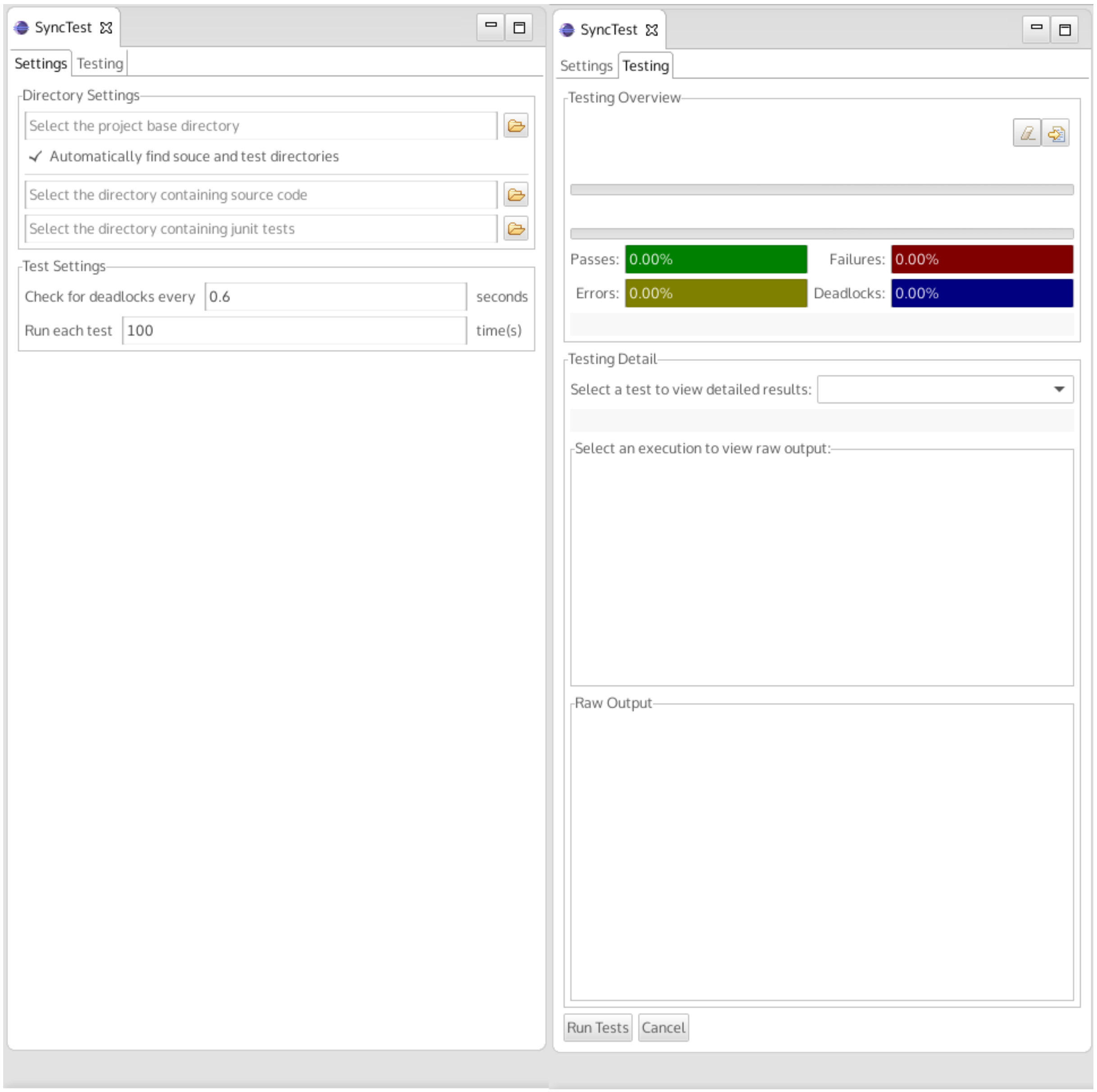
After I had finished working on the bash script I began work on a visual prototype of the plugin (Figure 7). This prototype would be a plugin created in Eclipse based on the JUnit plugin and my sketches. I decided to use a tabbed view with two tabs: one for configuring the plugin and one for displaying the test results. I got the idea for this from Eclipticon, which also uses a tabbed view. The settings tab contains 3 groups of settings for configuring the directories, setting input values for the script, and settings for the parser class. The directory settings are composed of 3 text boxes for entering paths to the source code, tests, and output directories. There are also buttons which spawn windows for selecting these directories. The testing group contains two text boxes for entering the number of times to run each test and the number of seconds to wait between deadlock checks. There was also a slider labeled ’pass threshold’ but it was ultimately removed because it no longer made sense within the context of the plugin. The final group containing parser settings was more of a placeholder in case I wanted to add some settings for the parser class. It was also eventually removed because the parser class did not require any specific configurations.

The testing tab is much more complicated, so I will go through it one element at a time from the top downward. The first element is a row of six boxes, which are buttons that I had intended to implement with similar functions to the JUnit plugin. The implementation of these buttons will be covered in the Major Changes section. Next is a progress bar and label which would show the user which test is currently running and the overall progress of all the test executions. Below the progress bar is four boxes containing the percent of test executions that passed, failed, deadlocked, or resulted in an error.

After this is the two bars that I had introduced in the section on the sketches. The first bar contains the overall test results: 70% of the bar is green for the tests that passed, 10% is red for the tests that failed, and 20% is blue for the tests that deadlocked. Any errors would show up in yellow. The second bar shows the same values but for the executions of a specific test instead of the overall results. In this case, 'sampleTest4' is selected and the bar is fully green, indicating that all of sampleTest4's tests passed. Next is a lists of the tests that can be selected to update the second result bar as well as the widget below which shows the failure trace for that test, should there be any failures. There are 3 buttons above the failure trace widget which were removed when the failure trace widget became a text box for the raw output of a test. Finally, at the very bottom there is a button to start the script and begin running tests.

5.4 Functional Prototype

The next step after creating the visual prototype was to add the functionality of the script to the visual prototype of the plugin. To do this, I modified the bash script so that it could accept inputs from the plugin interface, run the tests normally, and then print results back to the terminal. The plugin will read the output from the script and use it to update the user interface.



Directory Settings

Select the project base directory

✓ Automatically find souce and test directories

Select the directory containing source code

Select the directory containing junit tests

Test Settings

Check for deadlocks every 0.6 seconds

Run each test 100 time(s)

Testing Overview

Passes: 0.00% Failures: 0.00%

Errors: 0.00% Deadlocks: 0.00%

Testing Detail

Select a test to view detailed results:

Select an execution to view raw output:

Raw Output

Figure 8: SyncTest Current Version

The SyncTest plugin in its current states. The major changes listed in Section 5.5 are reflected here.

5.5 Major Changes

The plugin underwent many visual and functional changes through the course of its development. The most important changes are listed below. The visual changes can be seen in Figure 8.

- **Changes to directory settings:**

  The option to set an output directory was removed because the location of this directory didn't actually matter, it just had to exist. In addition, the option to select a project base directory was added and the plugin could attempt to automatically find the directories containing source code and tests. The output directory would be created in the provided base directory.

- **Second progress bar:**

  The visual prototype of the plugin only contained one progress bar for overall progress. A second progress bar was added so that the first bar would track the progress of all the tests and the second progress bar would track the progress of the executions of the currently running test.

- **Displaying raw output instead of failure trace:**

  Initially I had intended to display the failure trace for a selected test that had failed but instead I chose to show the raw, command line output for the test. This was easier to implement because all the output from the tests was already saved into text files. Also, if any tests did failed, their failure trace would be in the output file.

- **Changes to result boxes:**

  I coloured the result boxes to match their colours in the canvas to provide more clarity for the user. The result boxes can also be clicked to toggle between a fraction and percentile of the total executions.

- **Testing tab buttons:**

  I added two buttons to the testing tab: one to clear all of the widgets in the tab and one to export the current test results into a text file. I had intended to add some of the buttons that the JUnit plugin has but I did not have enough time.

- **Splitting testing tab into overall and detail groups:**

  The visual prototype for the plugin featured one large group in the testing tab which displayed all the results. In order to make it more clear which results were for overall testing results and which were for detailed results, the tab was split into two groups. One group would contain overall test results and one would contain detailed results for a selected test.

- **Removing Parser class:**

  In order to view the results of tests as they completed, I had to remove the Parser class and replace it with data classes for test results and execution results. This way the user can view details about tests as they finish instead of waiting until all of the tests are completed. Additionally, the Parser class would be very slow if there were a large amount of test executions and there would be a significant delay between the completion of the tests and the display being updated. Obviously, the removal of the Parser class eliminated this issue.

- **Java conversion:**

  The final and arguably largest change was the removal of the bash script that ran in the background. The bash script limited the plugin to the Linux-based operating systems, so it was removed and its functionality was replaced with Java code so that the plugin could (theoretically) run on any operating system.

5.6 Plugin Metrics

These metrics were generated by the Metrics 1.3.6 plugin for Eclipse. They are for the Java-only version of the plugin.

Table 1: Plugin Metrics

Metric	Package				
	synctest	synctest.views	synctest.testing	synctest.util	Total
Packages	N/A	N/A	N/A	N/A	4
Classes	1	1	3	2	7
Methods	3	12	12	15	42
Attributes	0	47	34	10	91
Lines of Code	25	675	307	106	1113

5.7 Case Study

To see how the plugin works, we will return to the bank account program from Section 4. Any references to the interface can be seen in Figure 8. The first step is configuring the inputs. SyncTest needs the base directory for the project being tested as well as the directories containing source code and JUnit tests. Alternatively, a user can select the base directory only and the plugin will attempt to locate the source and test directories automatically. Next the user needs to set the amount of time to sleep between deadlock checks and the number of times to run each test. After this the user can switch to the testing tab to begin running tests.

After the user pushes the ’run tests’ button, the tests will begin running and the progress bars will start filling up: the first one for overall test progress and the second for individual execution progress. As sets of tests are finished, they will appear in the drop-down menu in the testing detail group. The user can select a set of tests from this menu to view each of the executions and their result. The user can then select an execution and view it's raw output in the widget directly below. Once all of the tests have completed, the user can push one of the buttons at the top of the view to either save the results to a text file or clear all of the widgets to begin running tests again.

6 Conclusions

6.1 Summary

I began by introducing unit testing and concurrency and how they are not suited to work together. This lead to the motivation for this thesis: the development of multicore unit testing tools for Java. I then covered some related tools including JUnit and SyncDebugger, which were the foundation for this project. Next I introduced the first tool: a bash script for running JUnit tests and detecting deadlocks. This script provided the functionality that we desired but lacked a proper user interface, which leads to the Eclipse plugin. The Eclipse plugin maintains the same functionality as the bash script while also providing an interface for accepting inputs and displaying outputs. I went through the initial prototypes, explained some of the major changes, provided some statistics, and closed with a use case.

6.2 Contributions

To the field of unit testing multicore software we have contributed two tools. One is a bash script command line interface which runs JUnit tests multiple time while checking for deadlocks. While this tool satisfies our needs for functionality, it lacks a pleasant user interface. In response to this, we also created an Eclipse plugin. The plugin began as a front end for the command line tool until the bash script was removed and replaced with code in the plugin. Now we have a plugin that meets our functional and visual requirements. While these tools have met our requirements, they require further evaluation and improvements before they can be deployed as unit testing tools.

6.3 Limitations

The main limitations in this thesis lie within the bash script and the version of the plugin that runs a modified version of the bash script, which can only be run on Linux and Mac OS machines. The separate deadlock detection script is a C-Shell script which again, can also only be run on Linux and Mac OS machines. There are also limitations to the structure of projects that can be used with SyncTest. For example, SyncTest does not recursively compile or run tests, so all of the tests must be in one directory. Additionally, having each test in a separate file will provide more accurate results. The directory containing source code should also be structured the same way. In terms of scalability, SyncTest has only been tested with very small programs. It is my belief that much more testing needs to be done on programs of varying purpose and size before SyncTest can be considered a completed tool.

6.4 Future Work

For this plugin to meet the vision of the original proposal, more work must still be done. The main feature to be added is the incorporation of SyncDebugger. During the course of this thesis, I used SyncDebugger to instrument code for testing but I used it manually, outside of the plugin. Ideally, SyncDebugger should be run automatically when a user begins to run tests through SyncTest. A place-holder function was created to facilitate this addition.

The orignal vision for SyncTest was for it to be JUnit plus SyncDebugger and deadlock detection. JUnit has several useful features that need to be passed on to SyncTest to meet this vision. For example, JUnit allows you to re-run only the tests that failed; this would be very useful in SyncTest and could be extended to re-run only the tests that deadlocked. It would also be useful to have a right-click menu with an option to re-run a test a specified amount of times. JUnit also shows the amount of time each test took to run which can be viewed in the output files SyncTest creates, but ideally the times should be shown in one of the widgets. JUnit also has various options for viewing results such as showing only failures or only tests that were skipped. These are also useful features that SyncTest would benefit from.

7 Bibliography

[LSW07] Brad Long, Paul Strooper, and Luke Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3):281294, Mar. 2007

[U08] Ur, Shmuel (2008). Testing and Debugging Concurrent Software [PDF]. Retrieved from https://www.research.ibm.com/haifa/projects/verification/contest/papers/testingConcurrentJune2008ForMS.pdf