UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Tale of Two Testing Tools

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in
Computer Science

by

Robert Chang-che Chen

Committee in charge:

Professor William E. Howden, Chair
Professor Walter Burkhard
Professor Joseph Pasquale

2008

The Thesis of Robert Chang-che Chen is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT OF THE THESIS

A Tale of Two Testing Tools

by

Robert Chang-che Chen

Master of Science in Computer Science

University of California, San Diego, 2008

Professor William E. Howden, Chair

In today's software world, the increasing number of software products creates an equivalently increasing number of software bugs. Software engineers counter this by writing tests to check the functionality of their programs. However, as programs become increasingly complex, the process of writing tests becomes lengthened, tedious, and arduous. In response, certain products have been developed to automatically generate and run tests for software engineers. This study explores two such programs, Jtest and Agitator. Given a software program, these tools automatically generate and run tests for the programmer. This study attempts to show the effectiveness of these tools by feeding them sample programs, and observing the types of tests produced. At the end, the tradeoffs between these tools are compared.

I

Introduction

One of the goals of software testing is to build tools that can automatically generate and run tests and then validate results. In general, we know that it is not possible to generate tests that will find all bugs, but the goal is to be as effective as possible.

In this work we examine the capabilities of two commercial Java testing tools: Jtest and Agitator. Both are oriented to the testing of class methods, as opposed to systems testing. The tools will take a class C, and then test its methods m() by generating test data for C's class variables and m()'s parameters.

For each tool we consider: user interface features, test generation, oracles, test management, and static analysis.

With respect to test generation we consider:

1. Will the tool generate tests automatically for all types, including both primitive and non-primitive?

2. What is/are its test completeness measure(s)? e.g. branch coverage

3. Analyzing control flow which involves different types of data?

4. Complex control flow structure. What will it do if the method you are testing calls other methods?

5. Are we able to guess properties of the (proprietary) generation mechanisms from the examples we analyzed?

6. How are the tests specified? For example, can the tool generate JUnit test classes for its tests?

With respect to test oracles we considered:

1. Oracle construction

2. Manual test evaluation

In answer to the question "which tool is best" we hope the information will assist developers in making the best decision for themselves.

II

Jtest

Jtest is an Eclipse plug-in developed by Parasoft, which markets a suite of development tools. The user typically enters code into the integrated development environment (IDE) and clicks "Run" to perform testing, as shown in Figure 1 and Figure 2.

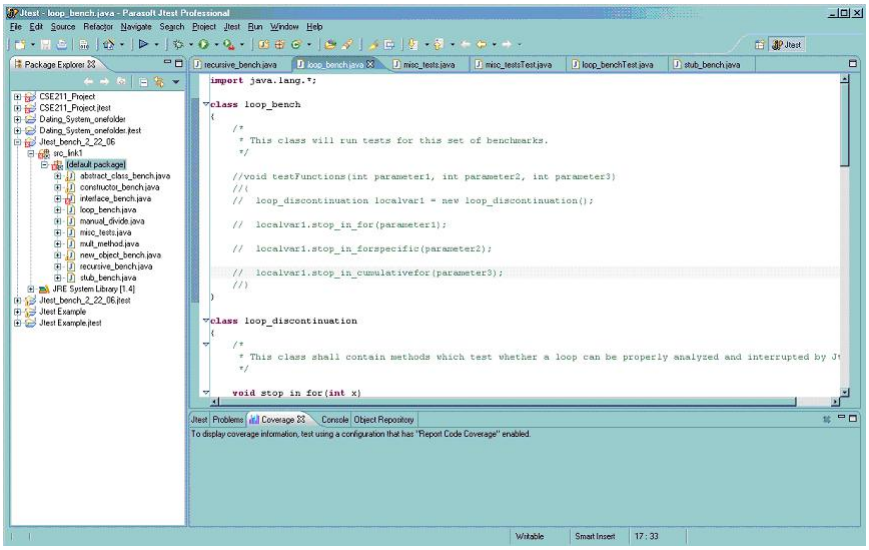

Figure II.1: The Eclipse IDE with Jtest plug-in.

Figure 2 shows the configuration menu. This allows control of coding standards to be checked; a choice of line or branch coverage; and which folders to

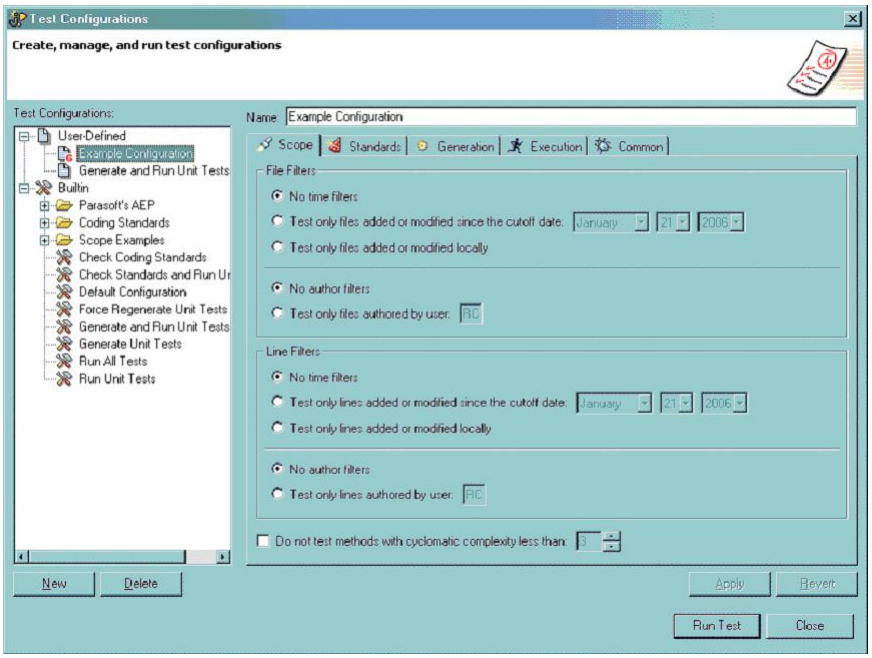

Figure II.2: The configuration menu with "Run Test" button to start testing.

execute from, among other features. When the testing is finished, the IDE will display a window stating the percentage of code covered, shown in Figure 3.

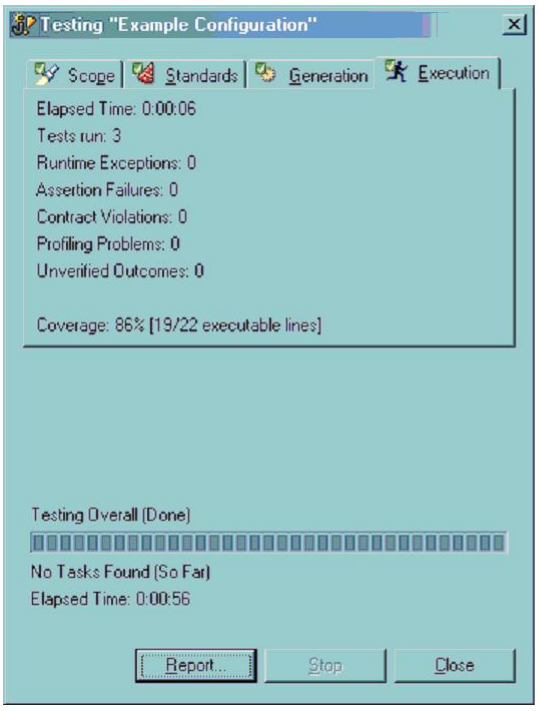

Figure II.3: Window displaying results of a test run.

It also generates a slightly more detailed HTML report which shows numbers of tests run and numbers of positive or negative outcomes. Jtest generates testing code via the JUnit testing framework. Each time a source file is tested, a file of JUnit test code is generated in a separate source tree, with file names "[originalfilename]Test.java". Figure 4 shows a typical JUnit test file.

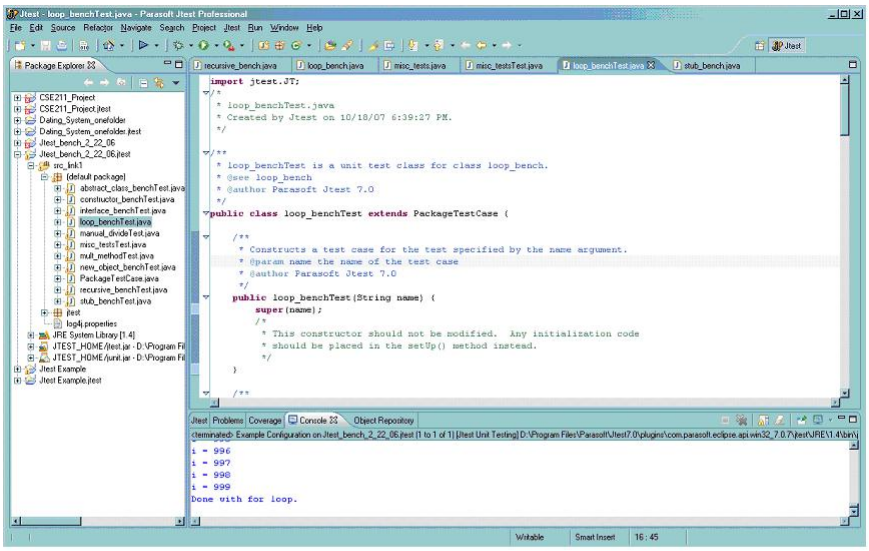

Figure II.4: A sample [filename]Test.java file.

Furthermore, the IDE displays red and blue marks for uncovered and covered lines of code, respectively, seen in Figure 5.

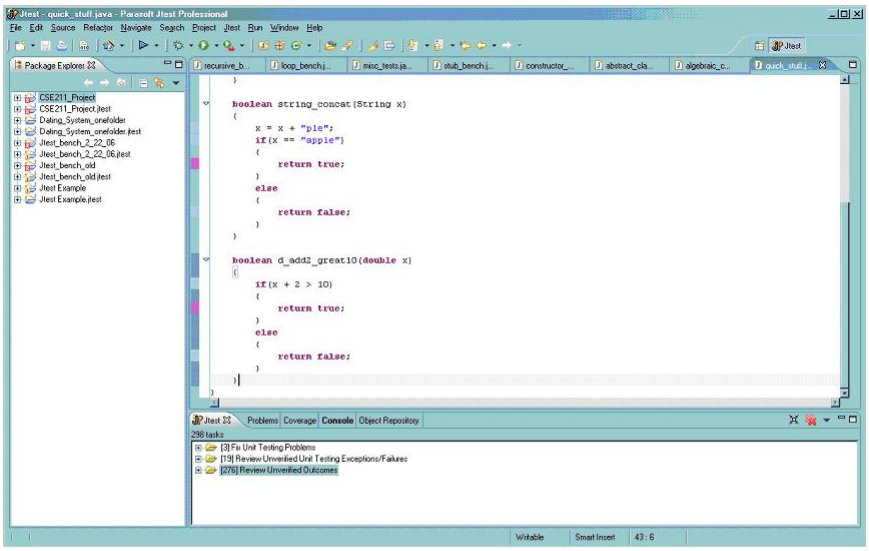

Figure II.5: Red and blue marks indicate uncovered and covered lines, respectively.

In the following, we will consider test generation, the oracle problem, test management, and static analysis.

II.1 Test Generation

II.1.A Types, Constructors, and Test Generation

Jtest can generate tests for all primitive types. The number and kinds of tests depends on its coverage attempts, which are discussed below. Java defines several primitive types, such as string, int, float, double, etc. Non-primitive types include user-defined classes. A method m() may have a non-primitive type parameter.

When a method m() from a class C is tested, we need to also consider the class variables for C and whether they are primitive. In addition, it is necessary

to consider if C has constructors, and if so, how many.

When a method m() in a class C is tested, the testing is preceded by a call to one of Cs constructors (if it has one). If C has no constructor, then Jtest will assign default values for the primitive type class variables. If C has a non-primitive class variable of type D, then Jtest is unable to generate instances of D. In general, Jtest does not fully specify what the instance of a containing class is for a test of a method, so it is necessary to do some analysis to determine what the default settings are for class variables when there is not a constructor. Figure 6 illustrates what occurs when no constructors are present.

```
class P{
        int a;
}

class Q{
        int i;
        P sample_P;
}

class PQ_Driver{
        boolean compare_intQ(Q inputQ){
                if(inputQ.i == 0){
                        return true;
                }
                else{
                        return false;
                }
        }

        boolean compare_intP(Q inputQ){
                if(inputQ.sample_P.a == 0){
                        return true;
                }
                else{
                        return false;
                }
        }
}
```

```
/**
 * Test for method: compare_intP(Q)
 * @throws Throwable Tests may throw any Throwable
 * @see PQ_Driver#compare_intP(Q)
 * @author Parasoft Jtest 7.0
 */
public void testCompare_intP1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("PQ_Driver"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("PQ_Driver"),
                THIS,
                "compare_intP",
                new Object[] { null },
                new Class[] { Q.class })).booleanValue();
        // NullPointerException thrown, originator is arg 1 to <Method
PQ_Driver.compare_intP(LQ;)Z>
        // at PQ_Driver.compare_intP(constructor_bench.java:113)
        // jtest_unverified
}
```

Figure II.6: An example of classes with no constructors, and generated test code.

Only the method "compare_intQ" is tested. The test for "compare_intP" fails because Jtest cannot generate an object which is a member variable of another object. Instead, a null object is generated. One can see from the Jtest output that the test for "compare_intP" failed because of a null pointer exception.

In the case where the method m() is in a class C that has a constructor, then Jtest will call the constructor and generate test values for both the constructor and the method parameters.

Subject to the above limitations, Jtest will generate a wide range of test values for different primitive types.

II.1.B Default Values

There are several kinds of default values. In the case discussed above, where a class C has no constructor, default values need to be assigned to class variables before proceeding with a testing. There is a single default value for each type:

int = 0

string = "0"

float/double = 0.0

When Jtest is testing a method m() it was observed to use the default values 0 for integers, 0.0 and 7.0 for float and double parameters, and "0" for strings.

Jtest will also choose additional kinds of relative "default" values when testing methods. If a method m() contains an integer constant k, then it will use k-1, k and k+1 for integer input method parameters. It does not do the same thing for float/analysis constants. For string constants k occurring in a method, it will use k as an input value for string input parameters.

II.1.C Coverage and Test Generation

According to Parasofts advertisement, Jtest, "[monitors] test coverage and achieves high coverage using branch coverage analysis." [Parasoft (2008)] In theory, it is impossible to know if all possible coverable branches have been tested by a test set T. Hence, it is necessary for Parasoft to use the cautious claim high coverage. However, in order to achieve high branch coverage, the tool should be performing some form of analysis. We did not know the inner workings of the tool, so the following comments were based on our own speculations, working backward from numerous examples.

Automated coverage analysis involves examining the conditional branches that occur along a path in the code, and then determining inputs that will cause that path to be followed. We considered different levels of complexity:

1. method control flow structure
2. types
3. expressions

Analysis and Control Complexity

Jtest is very proficient in performing analysis of "if" statements. There were no significant limitations in generating test data for "if" control blocks. Although failures occurred when dealing with complex arithmetic expressions and certain data types, we will address this issue in the next section.

An intriguing aspect of Jtest is its ability to handle "for" and "while" loops. Both are handled in the same manner, so examples presented will only feature the "for" loop. The analytical problem which comes to mind in dealing with loops is whether Jtest performs any analysis on a loop and make a prediction of what it will do. If the incrementing statement of a "for" loop is used as an input to a function, it is difficult for Jtest to generate a proper incrementing value.

As seen in Figure 7, Jtest is unable to cover the "if" statement because it only generates an input of a = 1000.

```
void incrementfortest(int a){
        int i;
        for(i = 0; i < 1000; i += a){
                if(i == 33){
                        i = 1000;
                }
        }
}
```

```
/**
 * Test for method: incrementfortest(int)
 * @throws Throwable Tests may throw any Throwable
 * @see loop_discontinuation#incrementfortest(int)
 * @author Parasoft Jtest 7.0
 */
public void testIncrementfortest1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("loop_discontinuation"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        JT.invoke(
                Class.forName("loop_discontinuation"),
                THIS,
                "incrementfortest",
                new Object[] { new Integer(1000) },
                new Class[] { Integer.TYPE });
        // No exception thrown
        // jtest_unverified
}
```

Figure II.7: An example of Jtest generating values for the incrementor in a loop.

Jtest also has difficulty determining a reasonable stopping condition for a loop. Figure 8 shows a situation where the stopping condition of a loop is an input to a function. Jtest was only able to generate 0 as the input. An input of 800 would have produced full coverage.

```
int stoppingfortest(int a){
        int i;
        for(i = 0; i < a; i++){ }
        if(i == 800){
                return 1;
        }
        else{
                return 2;
        }
}
```

```
/**
 * Test for method: stoppingfortest(int)
 * @throws Throwable Tests may throw any Throwable
 * @see loop_discontinuation#stoppingfortest(int)
 * @author Parasoft Jtest 7.0
 */
public void testStoppingfortest1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("loop_discontinuation"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        int RETVAL = ((Integer) JT.invoke(
                Class.forName("loop_discontinuation"),
                THIS,
                "stoppingfortest",
                new Object[] { new Integer(0) },
                new Class[] { Integer.TYPE })).intValue();
        assertEquals(2, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.8: An example of Jtest producing input for a loop stopping condition.

It is also interesting to see that certain complex "for" loops are fully covered by Jtest. In an experiment in which the initial value and incrementing value are modified, Jtest is able to perform some basic analysis on these values. Figure 9 shows an example where Jtest is able to generate a test value of 347, which is the value of "i" after the first iteration of the loop.

```
void hardestfortest(int x){
        int i;
        for(i = 342; i < 1000; i+=5){
                if(i == x){
                        i = 1000;
                }
        }
}
```

```
/**
 * Test for method: hardestfortest(int)
 * @throws Throwable Tests may throw any Throwable
 * @see loop_discontinuation#hardestfortest(int)
 * @author Parasoft Jtest 7.0
 */
public void testHardestfortest1() throws Throwable {
        Object THIS = JT.createObject(

Class.forName("loop_discontinuation"),
                        new Object[] {},
                        new Class[] {});
        // jtest_tested_method
        JT.invoke(

Class.forName("loop_discontinuation"),
                        THIS,
                        "hardestfortest",
                        new Object[] { new Integer(347) },
                        new Class[] { Integer.TYPE });
        // No exception thrown
        // jtest_unverified
}
```

Figure II.9: An example of Jtest fully covering a complex loop expression.

Yet, for a very simple program like the one seen in Figure 10, Jtest is unable to generate tests which cover all paths. In the case of Figure 10, only the value of 0 is generated for "x". A value of 342 would have been sufficient for full coverage.

```
void simplefortest(int x){
        int i;
        for(i = 0; i < 1000; i++){
                if(x == 342){
                        i = 1000;
                }
        }
}
```

```
/**
 * Test for method: simplefortest(int)
 * @throws Throwable Tests may throw any Throwable
 * @see loop_discontinuation#simplefortest(int)
 * @author Parasoft Jtest 7.0
 */
public void testSimplefortest1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("loop_discontinuation"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        JT.invoke(
                Class.forName("loop_discontinuation"),
                THIS,
                "simplefortest",
                new Object[] { new Integer(0) },
                new Class[] { Integer.TYPE });
        // No exception thrown
        // jtest_unverified
}
```

Figure II.10: An example of Jtest trying to cover a very simple condition within a loop.

Analysis and Types

Jtest was able to analyze paths and most mathematical expressions involving integers. Integer expressions involving exponentiation could not be analyzed, but this is an understandable limitation. Jtest also does not appear to carry out analysis for float/double precision. Consider Figure 11, where only the values 0.0 are generated and the tests only cover return value 3.

```
int test_fd(float x, double y){
        if(x < -1.0){
                return 1;
        }

        if(y > 10){
                return 2;
        }
        else{
                return 3;
        }
}
```

```
/**
 * Test for method: test_fd(float,double)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#test_fd(float,double)
 * @author Parasoft Jtest 7.0
 */
public void testTest_fd1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        int RETVAL = ((Integer) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "test_fd",
                new Object[] { new Float(0.000000f), new
Double(0.0) },
                new Class[] { Float.TYPE, Double.TYPE
})).intValue();
        assertEquals(3, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.11: An example using float and double types, and generated test code.

In the case of strings, Jtest is able to carry out very simple analysis. Here, in Figure 12, all branches are covered with test data of “string1”, “string2”, and “0”.

```
/**
 * Test for method: xy_eq_nested(java.lang.String,java.lang.String)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#xy_eq_nested(java.lang.String,java.lang.String)
 * @author Parasoft Jtest 7.0
 */
public void testXy_eq_nested2() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "xy_eq_nested",
                new Object[] { "string1", "0" },
                new Class[] { String.class, String.class })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}

/**
 * Test for method: xy_eq_nested(java.lang.String,java.lang.String)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#xy_eq_nested(java.lang.String,java.lang.String)
 * @author Parasoft Jtest 7.0
 */
public void testXy_eq_nested3() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "xy_eq_nested",
                new Object[] { "string1", "string 2" },
                new Class[] { String.class, String.class })).booleanValue();
        assertEquals(true, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

```
boolean xy_eq_nested(String x, String y){
        if(x == "string1"){
                if(y == "string 2"){
                        return true;
                }
                else{
                        return false;
                }
        }
        else{
                return false;
        }
}
```

Figure II.12: An example of string test generation.

Analysis and Expression Complexity

At some level, expression complexity limits the possible analysis for all types, regardless of the complexity of the control structure. This is to be expected, since there are no expression solution algorithms for higher order expressions, such as integer equations and inequalities of order higher than 4. Also, it may be non-economic to develop solution procedures for a data type such as strings, even when the expressions are limited to something relatively simple such as nested concatenation.

In the following we show examples of varying degrees of complexity for the three types: int, string and float/double. Each example is a limiting case, where the expression complexity blocked Jtest from producing tests needed to ensure branch coverage. In Figure 13, one can see that Jtest only generated a test value of 0 for the exponentiation example code. This does not cover all branches of evaluation.

```
boolean int_cubed_eq_1000(int y){
        if(Math.pow(y, 3) == 1000){
                return true;
        }
        else{
                return false;
        }
}
```

```
/**
 * Test for method: int_cubed_eq_1000(int)
 * @throws Throwable Tests may throw any Throwable
 * @see algebraic_complexity_bench#int_cubed_eq_1000(int)
 * @author Parasoft Jtest 7.0
 */
public void testInt_cubed_eq_10001() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("algebraic_complexity_bench"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("algebraic_complexity_bench"),
                THIS,
                "int_cubed_eq_1000",
                new Object[] { new Integer(0) },
                new Class[] { Integer.TYPE })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.13: An example of exponentiation with an integer type, and generated test code.

In Figure 14, one can see that Jtest only generated a null string as input into the "string_concat" method. Because this does not cover all the branches possible, it seems that Jtest is limited in its ability to generate tests for string input.

```
boolean string_concat(String x){
        x = x + "ple";
        if(x == "apple"){
                return true;
        }
        else{
                return false;
        }
}
```

```
/**
 * Test for method: string_concat(java.lang.String)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#string_concat(java.lang.String)
 * @author Parasoft Jtest 7.0
 */
public void testString_concat1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "string_concat",
                new Object[] { null },
                new Class[] { String.class })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.14: An example of string concatenation using the "+" operator, and generated test code.

In Figure 15, Jtest only generates a test value of 0.0 for the method. Not all branches are covered in this example. It appears that Jtest may have some limitations when attempting to generate tests for the double type.

```
boolean d_add2_great10(double x){
        if(x + 2 > 10){
                return true;
        }
        else{
                return false;
        }
}
```

```
/**
 * Test for method: d_add2_great10(double)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#d_add2_great10(double)
 * @author Parasoft Jtest 7.0
 */
public void testD_add2_great101() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "d_add2_great10",
                new Object[] { new Double(0.0) },
                new Class[] { Double.TYPE })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.15: An example of algebraic complexity using the double type, and generated test code.

II.1.D Generation Mechanism

Analysis Process

The details of how Jtest actually generates automatic test data are proprietary. There are several approaches to this problem, including symbolic evaluation [Howden (1977)] in conjunction with a standard inequality package, different forms of interval analysis [DeMillo and Offutt (1991); Offutt et al. (1999)], and convergence methods [Korel (1990)]. We examine some of the tests that were generated to see if it is possible to guess the process.

Because Jtest generates a set of default test values, it was important to determine whether achieving coverage was just a coincidence, or the product of performing some actual analysis. Recall that for an integer type variable, the default value is 0. In addition to the default value, Jtest generates a set of "boundary" values for a given program. Every analysis sample results in using standard boundary values which correspond to the constants. For a constant "k" in a program, the possible boundary values generated are k−1, k, and k+1. In the following example, a boundary value -1 is generated, in addition to -2. Figure 16 shows the results of this program.

```
/**
 * Test for method: int_add2_great_0(int)
 * @throws Throwable Tests may throw any Throwable
 * @see algebraic_complexity_bench#int_add2_great_0(int)
 * @author Parasoft Jtest 7.0
 */
public void testInt_add2_great_01() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("algebraic_complexity_bench"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("algebraic_complexity_bench"),
                THIS,
                "int_add2_great_0",
                new Object[] { new Integer(-1) },
                new Class[] { Integer.TYPE })).booleanValue();
        assertEquals(true, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}

/**
 * Test for method: int_add2_great_0(int)
 * @throws Throwable Tests may throw any Throwable
 * @see algebraic_complexity_bench#int_add2_great_0(int)
 * @author Parasoft Jtest 7.0
 */
public void testInt_add2_great_02() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("algebraic_complexity_bench"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("algebraic_complexity_bench"),
                THIS,
                "int_add2_great_0",
                new Object[] { new Integer(-2) },
                new Class[] { Integer.TYPE })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

```
boolean int_add2_great_0(int y){
        if(y+2 > 0){
                return true;
        }
        else{
                return false;
        }
}
```

Figure II.16: An example of simple integer type analysis, and generated test code.

In Figure 16, there are two constants in the program, 0 and 2. The possible boundary values choose from would be -1, 0, 1, 2, and 3. The default value for y would be 0. Jtest seems to choose test data from the set of boundary and default values. Yet, the choice of -2 demonstrates some sort of analysis was performed determine this unique test datum.

In order to see if the analysis goes beyond using default and boundary values, we constructed examples such that choosing the default and boundary values would not satisfy all paths. In Figure 17, which contains the condition x+5 < 37, Jtest generates values 31 and 32 and satisfies all paths despite our efforts to fool it.

```
/**
 * Test for method: easy_path(int)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#easy_path(int)
 * @author Parasoft Jtest 7.0
 */
public void testEasy_path1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        int RETVAL = ((Integer) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "easy_path",
                new Object[] { new Integer(31) },
                new Class[] { Integer.TYPE })).intValue();
        assertEquals(1, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}

/**
 * Test for method: easy_path(int)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#easy_path(int)
 * @author Parasoft Jtest 7.0
 */
public void testEasy_path2() throws Throwable {
                Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        int RETVAL = ((Integer) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "easy_path",
                new Object[] { new Integer(32) },
                new Class[] { Integer.TYPE })).intValue();
        assertEquals(2, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

```
int easy_path(int x){
        if(x + 5 < 37){
                return 1;
        }
        else{
                return 2;
        }
}
```

Figure II.17: An example of simple integer type analysis.

In this case, the default values of 0 and 37 would have traversed both logical branches, but Jtest seemed to perform analysis and used non-default and non-boundary values instead. A typical approach to this case would be to simplify the expression to x < 32, and then use boundary testing to determine the values 31, 32, and 33. The fact that the suggested values are so close to the actual values, leads us to believe that this is Jtests actual methodology. Now, consider Figure 18 in which there are two branches, x+5 < 37 and x+5 < 35. For the sake of saving space, the corresponding generated JUnit code is removed from Figures 18-20, and the results are described in text.

```
int harder_path(int x){
        if(x + 5 < 37 && x + 5 < 35){
                return 1;
        }
        else{
                return 2;
        }
}
```

Figure II.18: An example of multiple algebraic expressions with integer type.

In this case, Jtest generates input values of x as 30 and 29. This indicates that Jtest chooses boundary values for simple linear inequalities, and then checks them against the other forms for validity. However, in more complex cases involving the programmer denying Jtest of the use of its default and boundary values, interesting values are generated. Figure 19 is an example where default and boundary values with simple analysis do not work because the inequalities prevent the use of such values.

```
int deny_defaults(int x){
        if(x > 57 && x!=56 && x!=57 && x!=58){
                return 1;
        }
        else
                return 2;
}
```

Figure II.19: An example of an arithmetic expression with Jtest boundary values not allowed.

In Figure 19, the test values generated are 57, 58, and 102. The same 102 value is generated when a similar condition is used with if(x > 56 && x != 55 && x != 56 && x != 57). If the expression is increased to if(x > 305 && x != 304 && x != 305 && x != 306), Jtest generates 305, 306, and 402 as test values. This indicates that if Jtest is prevented from using boundary values, then it chooses a value at the "next hundred plus two". The same effect occurs with negative numbers. Even more interesting is if the programmer denies 402, Jtest tries 502, seen in Figure 20.

```
int deny_defaults(int x){
        if(x > 305 && x!=304 && x!=305 && x!=306 && x!=402){
                return 1;
        }
        else
                return 2;
}
```

Figure II.20: An example where all boundary values are denied, and 402 is denied.

For very large inequalities like if(x > 400,000,000) and with boundary value denial, Jtest fails to generate input in the form of 400,000,102. This results in a failure to test all logical branches. Branch coverage seems to work for an inequality like if(x > 900) with boundary value denial. However, if(x > 1000) with boundary value denial fails to achieve full coverage. This behavior indicates that Jtest is performing a search procedure which starts with normal default and boundary values, and then uses a predetermined set of values (e.g. +/- 102, +/- 202, . . . , +/-1002) as extreme backup test values in case all other values fail.

In general, the exact details of Jtests test data generation scheme cannot be determined without access to the source code. However, the evidence shows that it uses a simple boundary testing approach, with limited extreme bounds for emergency situations.

Infeasible Paths

The solver also seems to be able to deal with infeasible paths. Consider the following example in Figure 21, containing an assignment and branch condition.

```
int infeasible_path(int x){
        int y = 14588*x + 552 - x;

        if(y > 34){
                return 1;
        }
        else{
                return 2;
        }
}
```

```
/**
 * Test for method: INfeasible_path(int)
 * @throws Throwable Tests may throw any Throwable
 * @see quick_stuff#INfeasible_path(int)
 * @author Parasoft Jtest 7.0
 */
public void testINfeasible_path1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("quick_stuff"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        int RETVAL = ((Integer) JT.invoke(
                Class.forName("quick_stuff"),
                THIS,
                "INfeasible_path",
                new Object[] { new Integer(0) },
                new Class[] { Integer.TYPE })).intValue();
        assertEquals(1, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.21: An example of an infeasible path.

There is no integer solution to the "return 2" side of the branch. While the solver does not find a solution (which is expected due to soundness), it does not freeze or fail to terminate.

II.2 Oracles

After running tests, it is important to verify the results of test cases run. Any methodology for verifying the results of tests is called an oracle. Jtest has two main methods: the use of annotations, and the use of manual evaluation.

II.2.A Annotations

Jtest allows the user to insert annotations in the style of Javadoc comments. Among others, there exist pre-condition, post-condition, invariant, throws or exceptions, and assert expressions in the annotation syntax. In the case of pre-conditions, they are taken into account during the analysis used to generate test data. If a pre-condition exists in the code, Jtest will only generate test data which does not violate the pre-condition. Figure 22 shows an example where only values of "x" which are less than one are generated because the pre-condition must be satisfied.

```
/** @pre x < 0 */
boolean foo(int x){
        if(x > 3){
                return false;
        }
        else{
                return true;
        }
}
```

```
/**
 * Test for method: foo(int)
 * @throws Throwable Tests may throw any Throwable
 * @see annotation#foo(int)
 * @author Parasoft Jtest 7.0
 */
public void testFoo1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("annotation"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("annotation"),
                THIS,
                "foo",
                new Object[] { new Integer(-1) },
                new Class[] { Integer.TYPE })).booleanValue();
        assertEquals(true, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

Figure II.22: An example of Jtest generating test data which satisfies only the pre-condition, and generated test code.

In this case, Jtest generates a value of "-1" to satisfy the pre-condition of x < 0. Unfortunately, this pre-condition forces the function to run only one branch within its body.

Post-conditions function as oracles indicating a valid or invalid result in the console output. The previous example is modified with a post-condition, as shown in Figure 23. Jtest generates a warning in Figure 24 because a value of x = -1 was generated for the function foo(). This does not satisfy the post-condition of x > 9.

```
/** @pre x < 0 */
/** @post x > 9 */
boolean foo(int x){
        if(x > 3){
                return false;
        }
        else{
                return true;
        }
}
```

```
/**
 * Test for method: foo(int)
 * @throws Throwable Tests may throw any Throwable
 * @see annotation#foo(int)
 * @author Parasoft Jtest 7.0
 */
public void testFoo1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("annotation"),
                new Object[] {},
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("annotation"),
                THIS,
                "foo",
                new Object[] { new Integer(-1) },
                new Class[] { Integer.TYPE })).booleanValue();
        // jcontract.PostException thrown
        // at annotation.foo$dbc$post(annotation.java:5)
        // at annotation.foo(annotation.java:13)
        // jtest_unverified
}
```

Figure II.23: An example of Jtest post-condition usage, and generated test code.

Note that the warning tells the user which test case in which the post-condition failed. In Figure 24, the post-condition failed during the execution of "testFoo1". The Jtest user can then open the "annotationTest.java" file and find the "Foo1" test and observe its input. This feature can be quite helpful in tracking down code inconsistencies.

In addition, intermediate assertions could be inserted in methods in order to carry out intermediate validation. First, it is noted that the assertion is not a

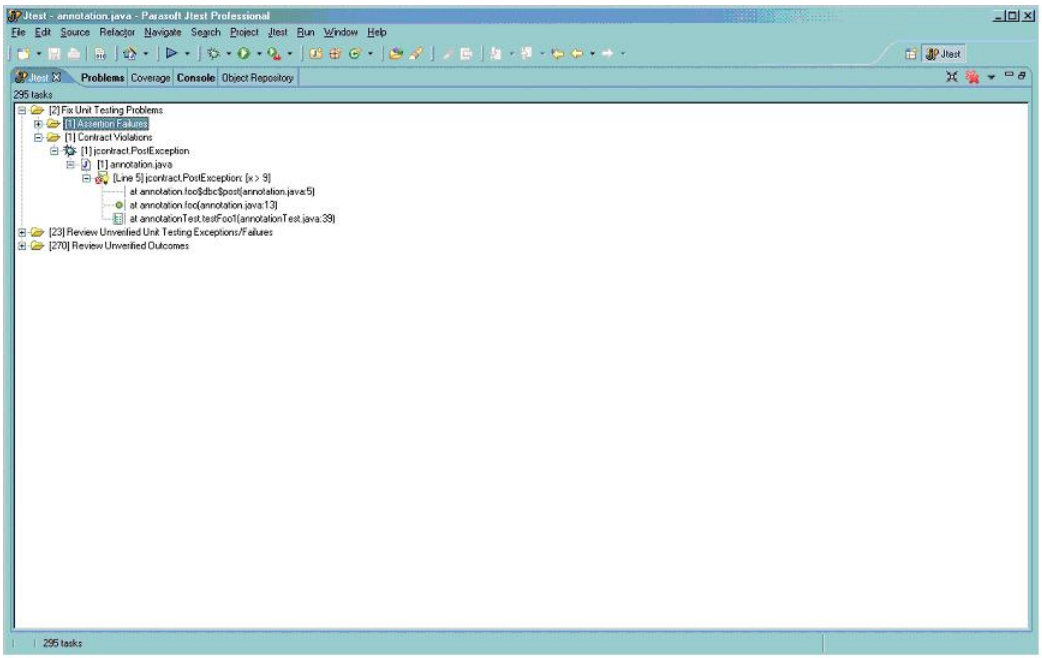

Figure II.24: The resulting warning of an unsatisfied post-condition.

general assertion that holds for all paths reaching some return value. It is a local assertion of the type assert(k = value), where "value" is a constant and "k" is a method input or class variable. The assertion is only valid at its position within the code. If the assertion fails, the rest of the code in the method is not executed in the test. Figure 25 illustrates these aspects of assertions.

```
/**
 * Test for method: foo(int)
 * @throws Throwable Tests may throw any Throwable
 * @see annotation#foo(int)
 * @author Parasoft Jtest 7.0
 */
public void testFoo1() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("annotation"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("annotation"),
                THIS,
                "foo",
                new Object[] { new Integer(0) },
                new Class[] { Integer.TYPE })).booleanValue();
        // jcontract.AssertException thrown
        // at annotation.foo(annotation.java:7)
        // jtest_unverified
}

/**
 * Test for method: foo(int)
 * @throws Throwable Tests may throw any Throwable
 * @see annotation#foo(int)
 * @author Parasoft Jtest 7.0
 */
public void testFoo2() throws Throwable {
        Object THIS = JT.createObject(
                Class.forName("annotation"),
                new Object[] { },
                new Class[] {});
        // jtest_tested_method
        boolean RETVAL = ((Boolean) JT.invoke(
                Class.forName("annotation"),
                THIS,
                "foo",
                new Object[] { new Integer(456) },
                new Class[] { Integer.TYPE })).booleanValue();
        assertEquals(false, RETVAL); // jtest_unverified
        // No exception thrown
        // jtest_unverified
}
```

```
class annotation{
        int y = 123;

        boolean foo(int x){
                /** @assert y > 122 */
                /** @assert x == 456 */
                if(x > 3){
                        return false;
                }
                else{
                        return true;
                }
        }
}
```

Figure II.25: An example of Jtest assertion usage, and generated test code.

In Figure 25, Jtest generates values of 0 and 456 for input x. The first assertion of y > 122 is always true because the class variable is defined as y = 123. The second assertion fails when the input x = 0. Hence, the "else" branch in the "foo" method is never executed. It is interesting to note that Jtest generates a test input of x = 456. This behavior indicates the use hard-coded values in the source as inspiration for test values.

Parasoft has another related product called JContract. JContract is able to analyze annotations and determine program feasibility at runtime. The mechanism which allows this feature is the use of Javadoc annotations within the code. Jtest is able to use these same annotations to create test cases.

II.2.B Manual Evaluation

Jtest provides the source files for JUnit code it generates, so one can observe the test cases being run. One JUnit test file is provided per source file being tested. However, the tool does not automatically provide final values of class variables used in testing.

A user interface is provided for defining stubs. It will also automatically generate generic stubs that return default type values for the I/O library and Enterprise Java Beans server side applications.

Finally, the user can generate his or her own tests with two methods. One can simply alter the JUnit testing code produced by Jtest, or define test objects in an object repository. A user can create any instance of a class with Jtests object repository. Given a class, the user can specify the values of class variables to any degree. The user would then run Jtest on the class with the user-defined objects created. In this case, Jtest would not automatically generate any data because the user has defined the data to be tested already.

II.3 Static Analysis

Because Jtest is an extension of the Eclipse IDE, the code is statically checked while the user types. It catches any errors deemed worthy by Sun [Sun (1999)]. These coding conventions help to avoid syntax errors, check for unused variables, and match file names to class names, among other conventions. Eclipses static checker allows for a more proper and standardized coding format.

II.3.A Rule Sets

Before running a test, the user may choose which advanced coding standards to follow in the test configuration window. Jtest provides 522 extra coding standards for users to choose from. These standards stem from the practices of other software engineers and books written on proper coding technique. There are also options to run tests based solely on the practices of certain software engineers and books.

Jtest documentation states that the user may also create user-defined rules. However, this is misleading because these user-defined rules can only be created from combinations of existing coding standards in Jtest. A user can choose which rules to use in a "rulesmap". Perhaps the most important use of user-defined coding standards is the ability to specify the severity of a rule. By lowering the severity of a rule, the user can essentially turn off the rule, and vice versa when increasing the severity.

II.3.B Management of False Positives

After the user has chosen a set of coding standards, Jtest will relentlessly check for violations of these standards. While the warnings given by this static checking do not harm the execution of the program, the user may be blindsided with multiple incursions of warnings, shown in Figure 26.

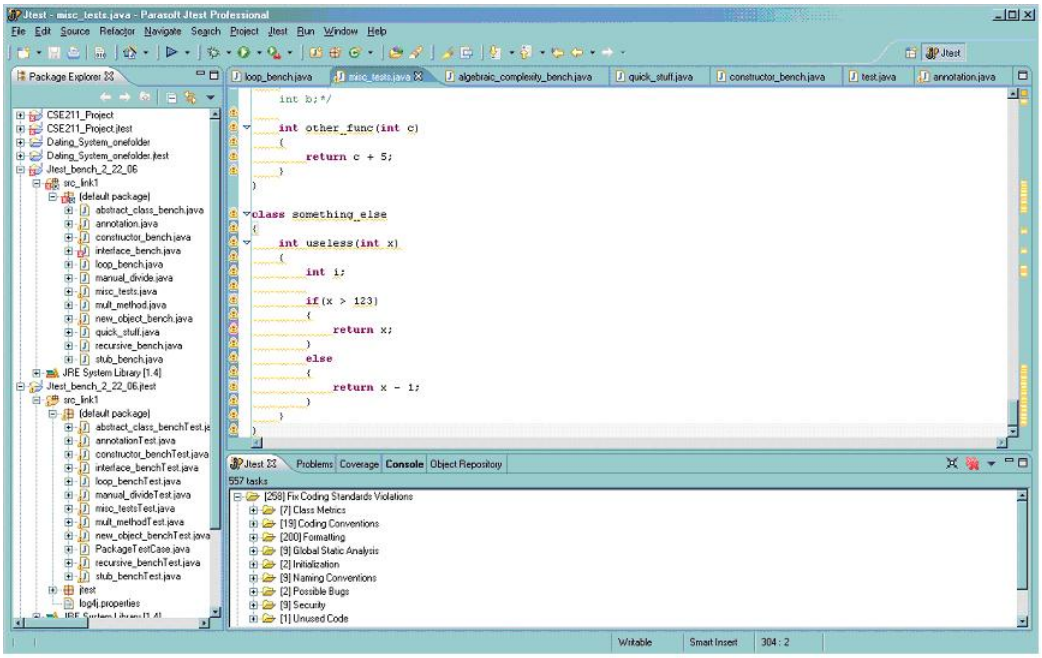

Figure II.26: An example of source code receiving multiple warnings per line.

Thankfully, Jtest allows for suppression of instances of warnings. In future runs, the same warning will no longer be displayed with the results of the test run. Instead, the suppressed warnings are saved elsewhere for the user to review.

III

Agitator

Agitator is a tool which has capabilities similar to those of Jtest, as well as additional novel features. We first consider the features discussed above for Jtest, in the same order, and then discuss some of its novel features. The tool is produced by Agitar, a new company founded in 2002.

One main difference from Jtest is Agitators test oracle. After any run of tests, Agitator does not generate a JUnit test file containing test cases or test input. Instead, Agitator provides a "snapshot window", which contains a sample of the test input generated and corresponding method results, seen in Figure 1.

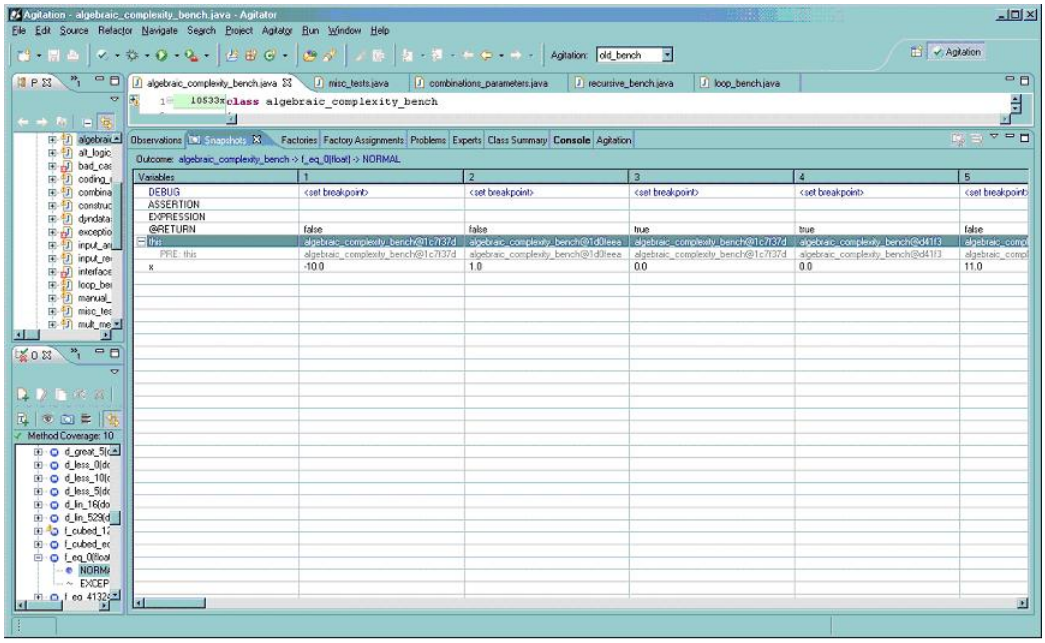

Figure III.1: Agitator "Snapshots" window.

Each "snapshot" window contains at most one hundred test cases generated by Agitator. Every test case presented is a test for a specific method within a class. The test cases are ordered one per column, and contain information such as the return value for that particular case; the state of the class variables; and the input values. For the sake of clarity, further references to snapshot windows will be in a tabular format, and not that of a screenshot.

III.1 Test Generation

III.1.A Types, Constructors, and Test Generation

Agitator is capable of generating tests for all the basic types. In the case where a method m() from a class C is tested, and C does not have a constructor, Agitator generates simple default values for class variables with primitive types. For the double type it is 0.0, for strings it is null, for char it is `\u0000`, and for integers it is 0. Figure 2 contains a code sample illustrating the behavior of a class with no constructor defined, and a subset of the output from Agitator tests. Only the path with a return value of 6 is ever reached. Note how the class variable values obtain default values for all test cases.

```
class undef_nocosntr_refinmeth{
        int a;
        int b;
        char class_char;
        String class_string;

        int ref_method(){
                if(a == 0 && b == 0.0 && class_char == '\u0000' && class_string ==
null){
                        return 6;
                }

                if(a == 88){
                        return 1;
                }
                else if(b == 99){
                        return 5;
                }
                else if(class_char == 'L'){
                        return 2;
                }
                else if(class_string == "black"){
                        return 3;
                }
                else{
                        return 4;
                }
        }
}
```

Variables	1	2	3	4	5	6
@RETURN	6	6	6	6	6	6
this.a	0	0	0	0	0	0
this.b	0	0	0	0	0	0
this.class_char	'\u0000'	'\u0000'	'\u0000'	'\u0000'	'\u0000'	'\u0000'
this.class_string	null	null	null	null	null	Null

Figure III.2: An example of a class with no constructor, but class variables are referenced.

If there exists a "set" method instead of a constructor, Agitator initiates the class variables with the default values. Set methods are identified in Agitator by any method name prefixed with the string "set*". Subsequent calls to the "set" method change the values of the class variables according to random Agitator input. The "set" method is used to create appropriate test environments for other methods. In Figure 3, the try_anything() method is fully tested with different class variable values. Note how the class variable values change with different test cases.

```
class undef_nocosntr_setinmeth
{
        int a;
        int b;
        char class_char;
        String class_string;

        int set_method(int x, int y, char z, String zz){
                a = x;
                b = y;
                class_char = z;
                class_string = zz;
                return 0;
        }

        int try_anything(int x, int y, char z, String zz){
                if(a == x){
                        if(b == y){
                                if(class_char == z){
                                        if(class_string == zz){
                                                return 1;
                                        }
                                        else
                                                return 2;
                                }
                                else
                                        return 3;
                        }
                        else
                                return 4;
                }
                else
                        return 5;
        }
}
```

Variables	1	2	3	4	5
@RETURN	5	3	5	5	4
this.a	-1	-1	-1	53	0
this.b	32	32	32	59	0
this.class_char	'3'	'3'	'3'	'\u0000'	'\u0000'
this.class_string	""	""	""	" Z"	Null
X	3	-1	6	2	0
Y	1	32	3	2	4
Z	'-'	'.'	'1'	'\u0002'	\u0004'
Zz	""	null	""	""	"$"

Figure III.3: An example of a "set" method changing the class variable values.

III.1.B Default Values and Random Tests

Like Jtest, Agitator chooses relative "surrounding" values based on constants appearing in a methods code. For example, if two hard coded values, 94833 and 832, are present in a method, Agitator generates 94833, 94832, 94834, 832, 831, and 833 as testing input. Figure 4 shows a subset of the generated tests for the example method.

```
boolean int_example(int x){
        int y = 94833;
        if(x > 832){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	True	False	False	True	True
X	833	832	-94833	890	833

Figure III.4: An example of "surrounding" default test values.

An interesting feature to note is that Agitator generates test values based on all hard coded values within a method. In the int_example() method, although the value 94833 has no influence on the control flow of the method, 94833 and its surrounding values were generated as test input. This may be because Agitator does not discriminate the usefulness or influence of variables when performing its analysis.

Agitator also uses random generation to construct test inputs. From the example in Figure 4, other random test values are generated such as 52, 6, -10, -100, and 925, among others. Unique numbers are also randomly generated. Numbers like pi and natural log are occasionally generated for the float and double types.

The user can specify one of three test generation modes: aggressive, extended and normal. These modes delineate the overall number of tests to run, with normal being the least number of tests, and aggressive being the most. Larger numbers of tests add a stress testing element to Agitator, in the sense that when a method is tested many times, it will stress the retained state properties.

Agitators string type input generation seems to be performed in the same manner as integer, float, and double types. For hard coded strings in a method, Agitator generates the same hard coded strings as test input, shown in Figure 5.

```
boolean string_cat(String x){
        if(x+"le" == "apple"){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	False	False	False	False	False
X	“abcdefghijklmnopqrstuvwxyz”	null	“”	“apple”	“le”

Figure III.5: An example of string concatenation with the “+” operator.

In the string_cat() method in Figure 5, Agitator generates strings of “le” and “apple” as test values. It also generates null and other random strings like “`abcdefghijklmnopqrstuvwx{z`” and “`CfO~z B7SnV(K8Bt\$%vEr^N^s}`”. It does not appear that Agitator can determine that the “correct” input to this string concatenation method is “app”. For a similar string, the concat() method is used to yield the same input values in Figure 6. Again, the “correct” input values are not generated.

```
boolean string_concat(String x){
        if(x.concat("le") == "apple"){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	False	False	False	False	False
X	“pple”	“”	“apple”	“le”	“2”

Figure III.6: An example of string concatenation with the “concat()” method.

III.1.C Coverage and Test Generation

In the advertising for Agitator, the vendor states:

To really unit-test code, every line, every branch, and every outcome must be tested. Thats a daunting combinatorial problem. Its not practical to create such thorough tests manually. The test code is usually longer than the code being tested, and time spent writing it is a direct tradeoff against time spent implementing necessary features. Agitator automatically creates dynamic test cases, synthesizes sets of input data, and analyzes the results.

The implication is that Agitator carries out analysis to determine data that will cause branches in the code to be tested on at least one test. Analysis of the tests that are generated indicates that there are special values specific to the coverage of branches that do not look like default or random tests.

Analysis and Control Complexity

As in the case of Jtest, the tool can deal with complex control structures during test generation. Agitator seems to traverse possible conditional branches by looking at values in conditional statements and generating test data to satisfy the conditions. For the large set of "if" conditions in Figure 7, Agitator is able to generate all the possible combinations of integers which result in all possible return values.

```
int intint_8_nest_IF(int a, int b, int c, int d, int e){
        if(a < 10)
                if(b > 1000)
                        if(c > 500)
                                if(d < 43)
                                        if(e > 432)
                                                return 1;
                                        else
                                                return 2;
                                else
                                        return 3;
                        else
                                return 4;
                else
                        return 5;
        else
                return 6;
}
```

Variables	1	2	3	4	5
@RETURN	5	10	10	9	9
this.a	-3	3437	501	-904	9
this.b	3436	3437	4	432	100
this.c	501	3437	4	432	8
this.d	8	3437	4	432	8
this.e	433	3437	501	-904	4
this.f	33	3437	4	-5	-8
this.g	-3	3437	501	4	9
this.h	999	3437	4	432	8
this.i	7	3437	4	432	8

Figure III.7: An example of nested “if” logic.

If one considers the “e” value in Figure 7, Agitator generates “e” as: 432, 433, -432, 1, 9, 1000, and more. Various other values are generated for the other input variables to the function. Agitator is able to generate input which reaches every branch in the method.

Agitator is also able to generate test cases for complex “for” loops. Figure 8 shows a method whose input is the stopping condition of a “for” loop. The loop increments a variable “y” by 2, and later, “y” is checked for a certain value, 41. In some test cases, Agitator is able to generate the correct stopping condition of x = 20.

```
void stopping_loop(int x){
        int y = 1;
        for(int i = 1; i < x; i++){
                y += 2;
        }
        if(y == 41){
                return;
        }
}
```

Variables	1	2	3	4	5
X	100	41	1	-41	20

Figure III.8: An example of “for” loop stopping condition generation.

The value of x = 20 seems to be generated occasionally. This is most likely because it is a random value generate by Agitator for the purpose of robustness testing, and not a standard value generated by analysis. As a result, it takes several runs for Agitator to achieve coverage of the if(y==41) line.

Another interesting loop test is to make the incrementor value an input to the method. Agitator seems to handle this well also. Figure 9 shows a "for" loop whose incrementor value needs to be 51 in order to achieve full coverage. Agitator seems to be able to obtain full coverage of this method consistently by generating x = 51 during some test runs. It is not clear whether this coverage is due to analysis or Agitators own random input generation process.

```
void incrementor_loop(int x){
        int i;

        for(i = 0; i < 103; i += x){
        }

        if(i == 5253)
                return;
}
```

Variables	1	2	3	4	5
X	-5253	5253	1	341	51

Figure III.9: An example of “for” loop incrementor generation.

For all normal "if" logic and most normal loop logic, Agitator performs well and attains full coverage. It is not clear whether Agitator is performing analysis on loops because full coverage is not always attained when a method contains a loop stopping condition as input. In the case of loop logic, we use "for" loops as the primary example. "While" loops also exhibit the same behavior, but it would be redundant to show those examples as well.

Analysis and Types

Agitator was able to determine coverage oriented tests for not only integers, but also floating point, and doubles. In Figure 10, one can see nested "if" statements with double and float types required. All branches were reached for the example in Figure 10.

```
double dd_8_nest_IF(double a, double b, float c, double d, float e){
        if(a < 10.973)
                if(b > 1000.823)
                        if(c > 500.623)
                                if(d < 43.5)
                                        if(e > 432.14463)
                                                return 1;
                                        else
                                                return 2;
                                else
                                        return 3;
                        else
                                return 4;
                else
                        return 5;
        else
                return 6;
}
```

Variables	1	2	3	4	5
@RETURN	4.0	5.0	6.0	5.0	5.0
This.a	-1000.0	3.0	432.0	2.0	9.973
This.b	1032.5728872527216	3.0	432.0	6.0	432.0
This.c	-1000.0	11.0	499.0	500.0	34.0
This.d	-5.0	-904.0	3.0	6.0	4.0
This.e	903.0	-100.0	499.0	42.42	-904.0

Figure III.10: An example of double and float types in a nested “if” structure.

Analysis and Expression Complexity

Agitator was able to deal with more complex expressions than Jtest. For integer expressions occurring in branching conditions, it could solve for values for involving exponentiation, which is a level above Jtests capability. The tool could also perform analysis involving floating point and double precision expressions. The example in Figure 11 shows that Agitator can solve complex algebra and expressions as well.

```
boolean d_lin_529(double y){
        if(y*7 + 3 == 529){
                return true;
        }
        else{
                return false;
        }
}
```

Variable	1	2	3	4
@RETURN	False	false	true	True
Y	42.96307275795496	3.0	75.14285714285714	75.14285714285714

Figure III.11: An example of algebraic complexity with float and double types.

Agitator was able to generate tests to cover all branches for the example in Figure 11. Somehow, it is able to solve a linear equation involving a double type. Figure 12 shows the capability for determining a basic cube root for a float type. All branches in Figure 12 were covered.

```
boolean f_cubed_eq_1000(float x){
        if(Math.pow(x, 3) == 1000){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	False	False	False	False	False
X	520.0	1.0	0.0	-899.0	5.0

Figure III.12: An example of algebraic complexity with exponents.

Agitator is also able to generate interesting data for methods without explicit constants. Figure 13 shows a method whose conditional statements rely purely on input variables with no constants within the code. Both branches are covered in this example.

```
int many_vars(int x, int y, int z, int Q){
        if(x+y*z > Q){
                return 1;
        }
        else{
                return 2;
        }
}
```

Variables	1	2	3	4	5
@RETURN	2	1	2	1	1
X	2	2	1	-5	-100
Y	-5	2	1	-5	0
Z	4	2	0	1	5
Q	832	2	1	-100	-124

Figure III.13: An example of an expression relying purely on variables and no constants.

Limitations

Some expressions and situations are difficult for Agitator to cover. For example, the very complex polynomial expression seen in Figure 14 involves taking the cube of "y" and dividing the result by the "y". The "true" branch is never covered in various Agitator test runs of the code in Figure 14.

```
boolean int_divx_great_999999(int y){
        if(Math.pow(y, 3)/y > 999999){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	False	False	False	False	False
Y	890	1	0	100	522

Figure III.14: An example of a complex polynomial expression uncovered by Agitator.

If a value of 1000 or greater were generated for "y", then all branches of this method would have been evaluated. However, no values of 1000 or greater are generated in all of the runs attempted. It seems that Agitator encounters a numerical limitation, generating values less than 1000, when dealing with polynomials. Furthermore, it seems that Agitator does not attempt to solve expressions involving polynomials. In Figure 15, only the precise value of 500 for "y" will cause the code to evaluate the "true" branch. This behavior is expected because automatically solving polynomial expressions is quite difficult for mere humans.

```
boolean int_sq_250000(int y){
        if(Math.pow(y, 2) == 250000){
                return true;
        }
        else{
                return false;
        }
}
```

Variables	1	2	3	4	5
@RETURN	False	False	False	False	False
Y	4	1	0	-2	890

Figure III.15: An example of Agitator struggling to solve exponentiation.

III.2 Observations and Oracles

Like Jtest, Agitator also contains the normal Eclipse-based oracle of green and red highlighted lines of code, respectively indicating covered and non-covered lines. However, Agitator provides a new oracle mechanism called "observations". These observations are guesses at simple relationships about class variables and method outcomes after each set of test runs. For example, Agitator can observe that a certain integer class variable is always equal to five, or the method foo() always returns a value of six. Agitator displays statistics on how often each observation was true or false. After each set of test runs, the user can decide to promote any observations into assertions which will be automatically checked on subsequent runs. Although these observations may not conform to a real functional oracle, they may perform the useful function of assisting the tester in understanding the code under test.

III.2.A Types of Observations

Agitator can make observations about class variables and methods. For class variables, Agitator attempts to determine the class invariants by guessing the typical values of class variables. Figure 16 contains a class with a constructor and a method which alters the class variables. After a run of tests, involving automatic test case construction, the "observations" in the table are produced.

```
class test_class{
        int a;
        float b;
        double c;
        int d;
        char x;

        test_class(int aa, float bb, double cc, char xx){
                a = aa;
                b = bb;
                c = cc;
                d = 12345;
                x = xx;
        }

        void alter_vars(int i, float j, double k, char l){
                a += i;
                b -= j;
                c *= k;
                x += l;
        }
}
```

alter_vars()			
	Value	True	False
1	this.d == 12345	1827	0
2	'\u0000' <= this.x <= '\'	1827	0
3	-1.8300087599665044E16 <= this.c <= 9.170147243831789E13	1827	0
4	-12347 <= this.a <= 12426	1827	0
5	-12362.369F <= this.b <= 12561.532F	1827	0

Figure III.16: An example of basic class variable observations.

For the class variables “a”, “b”, “c”, and “x”, Agitator generates a variety of values and observes that each variable could lie between ranges of possible values. The variable “d” remains constant after the instantiation of the class and is observed to have a relatively unchanging value of 12345. The “True” and “False” statistics indicate the number of tests for which an observation was true or false.

In addition to class observations, methods have their own individual observations. Namely, Agitator tries to determine the values of inputs to methods, the values of class variables before and after a method is run, the return value of the method, and the return expression of a method. Figure 17 shows these various observation types for a method that performs simple comparisons of class variables.

```
class test_class{
        int a;
        float b;
        double c;
        int d;
        char x;

        test_class(int aa, float bb, double cc, char xx){
                a = aa;
                b = bb;
                c = cc;
                d = 12345;
                x = xx;
        }

        int simple_method(int i, float j, double k){
                if(a == i){
                        if(b > j){
                                if(c < k){
                                        return 1;
                                }
                                else
                                        return 2;
                        }
                        else
                                return 3;
                }
                else
                        return 4;
        }
}
```

simple_method()			
	Value	True	False
1	-12345 <= @PRE(this.a) <= 12457	949	0
2	-12345 <= i <= 13306	949	0
3	-12345.0F <= @PRE(this.b) <= 12440.494F	949	0
4	-4.436025056432724E15 <= @PRE(this.c) <= 3.3215518154990392E18	949	0
5	-4.436025056432815E15 <= k <= 3.6361175908822359E18	949	0
6	-551.7054F <= j <= 13625.859F	949	0
7	1 <= @RETURN <= 4	949	0
8	@PRE(this.a) == i	672	277
9	@PRE(this.b) > j	495	454
10	@PRE(this.c) < k	496	453

Figure III.17: An example of basic method observations.

Observations 2, 5, and 6 describe observed properties of the test input parameter values for the simple_method(). These numbers are generated solely by Agitator without programmer alteration. Observations 1, 3, and 4 are ranges of values for the class variables before the method was run for each test. These values are determined by the parameters to the constructor upon the instantiation of each test, and by other methods whose test execution could result in changing the values. Observation 7 is the range of return values of the simple_method() call. Notice that observations 1 through 7 all contain no false results from tests run; this is because Agitator is merely stating the hard values of the tests it generated. In contrast, observations 8 through 10 have some counts of false tests observed. These observations are more generalized, and attempt to guess the relationships between the input parameters and class variables. Each of these observations is stated as a general expression, which leads to a mixture of true and false test cases.

III.2.B Using Observations

The value of Agitators observations is twofold. It provides insight into the kinds of tests that have been run. In addition, if the programmer determines that an observation is a correct invariant, the programmer can promote it to an assertion to be checked upon each test run. Figure 18 shows certain observations promoted. Simple_method() is rerun to determine whether the assertions hold true in testing.

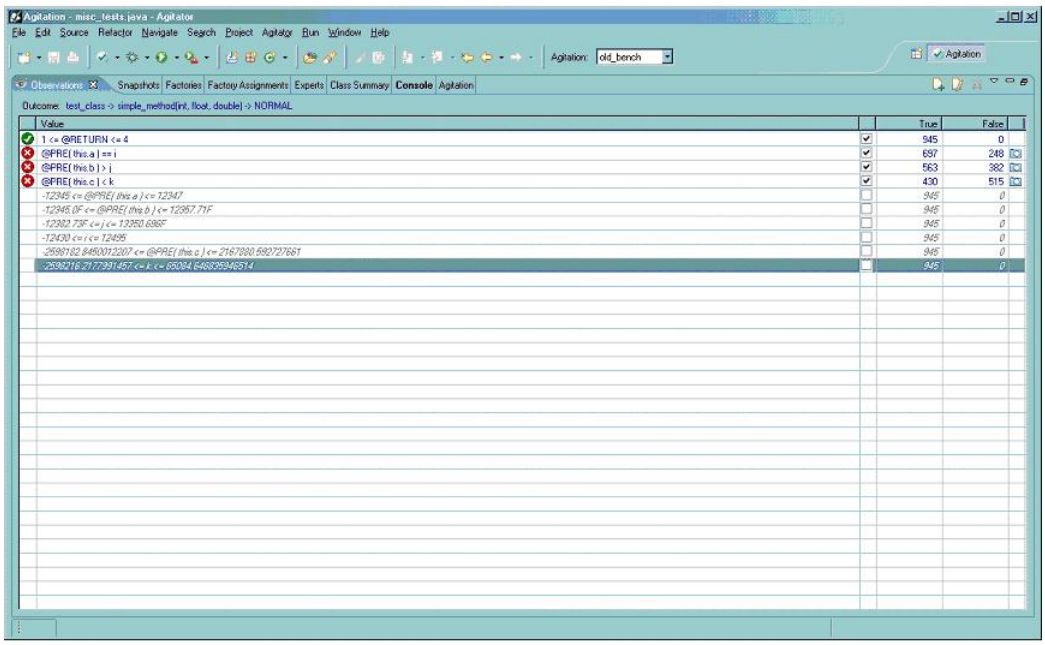

Figure III.18: An example of promoting observations to assertions using the code example in Figure 17.

The observations promoted to assertions are distinguished by check marks. One can see that the return value assertion is confirmed to be true (green) on test runs. This is rather trivial because simple_method() can only return 1, 2, 3 or 4, so the range of return values must be between 1 through 4, inclusive. The assertions made about the relationships between the class variables and input parameters are confirmed as false (red). Again, this result is expected because this example does not restrict the input of the method to conditions of a == i, b > j, or c < k. Note that a camera icon appears to the right of the assertions which failed. Clicking on this camera icon allows the user to see snapshots of test cases which caused the assertion to fail.

The usefulness of some of Agitators observations is doubtful. A user may not need or care to know the range of input values, or return values of a method. In this case, the user can simply create new assertions using a special syntax and assertion editor. In fact, Agitators automatically generated assertions can also be altered in an editor.

III.2.C Advanced Observations

When exploring more complex programs, Agitators ability to generate useful observations becomes more limited. To begin with, a relatively simple mathematical expression is shown in Figure 19, and produces interesting observations from Agitator.

```
class math_class{
        int expression(int x){
                return x - 2*x + 5;
        }
}
```

expression()			
	Value	True	False
1	@RETURN == ((x - (2 * x)) + 5)	527	0
2	@RETURN == (5 - x)	527	0
3	-95 <= @RETURN <= 105	527	0

Figure III.19: An example of a simple mathematical expression.

Figure 19 shows two interesting return value observations. Observation 1 is the original expression given directly in the return statement code, while observation 2 is an algebraically simplified version of the return statement code. The presence of observation 2 shows that Agitator is performing some sort of analysis in an attempt to create useful observations. The next example in Figure 20 complicates the expression.

```
class math_class{
        int mult_two(int x){
                x *= 2;
                return x;
        }

        int another_expression(int x){
                return x - mult_two(x) + 5;
        }
}
```

another_expression()			
	Value	True	False
1	@RETURN == ((x - @PRE(this.mult_two(x))) + 5)	707	0
2	@RETURN == (5 - x)	707	0
3	-95 <= @RETURN <= 105	707	0

Figure III.20: An example of observations for dependent methods.

The table in Figure 20 describes the observations for the method another_expression(). Again, the 5-x return value is seen in observation 2, but the other return value in observation 1 is different. Observation 1 indicates that the mult_two() method must be called prior to executing the another_expression() method. While this statement is true, it is not the most optimal or human-readable observation which could be made. Observation 1 in Figure 20 is a more obfuscated version of observation 1 in Figure 19. Yet, in both Figures, the most simplified observation of 5-x is identified.

There are a couple interesting features of the observations seen in Figures 19 and 20. Observation 3 in each of the figures shows that Agitator constructs observations for raw output values. It attempts to guess the actual numerical values which will be returned. Furthermore, Agitator seems uses any "return x" statement as a template for generating the corresponding method observation. In observation 1 of Figure 20, Agitator is unable to determine that mult_two(x) is actually the expression x*2, and merely inserts @PRE(this.mult_two(x)) into the observation. Although this observation is correct, it is not entirely useful to the programmer. Next, in Figure 21, the same code with one more added layer of complexity produces fewer observations than before.

```
class math_class{
        int mult_two(int x){
                x *= 2;
                return x;
        }

        int yet_another_expression(int x){
                x -= mult_two(x) + 5;
                return x;
        }
}
```

yet_another_expression()			
	Value	True	False
1	@RETURN == (5 – x)	708	0
2	-105 <= @RETURN <= 95	708	0

Figure III.21: An example of observations for a more complex dependent method.

The table in Figure 21 describes the observations for the yet_another_expression() method. The "return x" line adds one more level of indirection between the return statement and the target mathematical expression by assigning the expression to its own variable "x". This results in one fewer observation generated by Agitator, but still yields the important simplified form of the return value, 5-x. One can begin to see here that adding a level of indirection decreases the number of observations Agitator can make. In Figure 22, it becomes apparent that using a layer of indirection with multiple inputs results in useless observations.

```
class math_class{
        int mult_two(int x){
                x *= 2;
                return x;
        }

        int more_complex_expression(int x, int y){
                x += mult_two(y);
                return x;
        }
}
```

more_complex_expression()			
	Value	True	False
1	-100 <= y <= 100	705	0
2	-299 <= x <= 199	705	0
3	-300 <= @RETURN <= 300	705	0

Figure III.22: An example of observations with method dependencies and multiple inputs.

From Figure 22s observations, like that of the more_complex_expression() method, there is no return expression observation seen in the previous Figures 19 through 46. In addition to the "return x" level of indirection, there is the added complexity of calling a function on a second input parameter "y". It seems like Agitators ability to create return expression observations appears to be limited to one input parameter.

III.2.D Test Management

One of the nicer features of Agitators oracle, is that it shows you the test cases (at most 100) in detail. For every test it shows you the inputs and results from the method, as well as the final of the class variables. As in the case of Jtest, users of Agitator are presented with coverage descriptions. It also tells you how many times it ran each line of code, which goes beyond simple coverage indicators.

Agitator, is not a JUnit-oriented tool. It does not appear to generate observable files of JUnit classes for running future tests. The point of view taken in Agitator is if you exit a session, it will regenerate the tests for a subsequent session. This may be inefficient for large applications with many tests. However, users can define and save mock objects for further reuse.

Mock objects are instances of classes in which the user defines the values of class variables. The user can command Agitator to use the mock objects as test cases in addition to the automated tests generated by Agitator.

III.3 Static Analysis

In addition to Eclipses compile-on-the-fly checker, Agitator contains 138 coding rules for the programmer to use. Most of the built-in code rules check for coding style and format. The programmer can also create his or her own code rules using the CheckStyle 4.1 API. This is an open-source, Java code rules checker with

its own rule specification language. Agitator does not make any claim to a source or reference for the origins of their built-in code rules. However, a look at the CheckStyle website shows a great similarity between default CheckStyle rules and Agitators built-in rules.

Upon agitating the code, the static checker checks all lines of code for violations of the selected rules. When the agitation is finished, rules violations are reported in the Eclipse IDE as error marks beside the corresponding lines of code. Additionally, each violation is reported in a separate Experts window where the programmer can double-click on the violation and the IDE displays the line of code in violation.

IV

Summary and Conclusions

In this section, the similarities and differences between Jtest and Agitator are laid out. The benefits and drawbacks of each tool are considered, and a final opinion is offered on the preferred tool.

IV.1 Commonalities

Jtest and Agitator seem to have the same types of overall testing harness design. Each tool is implemented as a plug-in for the Eclipse IDE. The user writes code in the IDE and can observe the static checker marking violations of coding standards on-the-fly. Then, the user activates the testing mechanism to generate test cases and run the code with the generated input. Both tools show whether each line of code was covered during testing; lines marked green are covered, while lines marked red are not covered.

Another important similarity is their method for generating test data. Each tool has default values for primitive data types. Furthermore, each tool utilizes the hard-coded constants within code to generate boundary test cases. If a constant "k" is present in the users code, both Jtest and Agitator generate test inputs of "k+1" and "k-1", hoping to traverse all possible branches.

IV.2 Differences

The test oracles for Jtest and Agitator differ significantly. Jtest provides the complete JUnit code generated for all tests. This sort of documentation provides the exact input and output for all tests. These tests are saved and organized into Java files; one file is generated for every file tested. In contrast, Agitator only provides a snapshot of up to one hundred test inputs and results. These snapshots can only be seen in the Eclipse IDE, and are not saved in files. Snapshots only represent a portion of the actual number of tests generated. Typically, Agitator generates thousands of tests.

In the static analysis arena, Jtest provides more options than Agitator. Jtest is able to check 522 different built-in coding rules, while Agitator only contains 138 built-in coding rules. However, Agitator allows the user to create new coding rules with the CheckStyle syntax.

Agitator seems to fare better in testing data types and complex expressions. Jtest is unable to generate meaningful test data for float, double and string types, while Agitator succeeds admirably. Additionally, Agitator successfully covers all branches for certain expressions containing polynomials. Yet, Jtest only generates default values for polynomial expressions, resulting in poor coverage of branches.

A further advantage in Agitator is its numerous and repeated generation of random input for each run. During a run, Agitator performs a kind of stress testing which repeatedly calls methods with random input. A method can be tested nearly two thousand times in one run. Each run is different because new random values are generated every time. This helps in determining the robustness of code. On the other hand, Jtest has no such facility. In every run of Jtest on a specific piece of code, the user will receive the same set of tests every time. The set of tests is also much smaller; only enough to obtain the maximum amount of branch coverage possible. It does seem that stress or robustness testing is part of

Jtests features.

Finally, Agitators observations are arguably its most novel contribution to the tool. Agitator creates observations to show the user certain "invariants" about the behavior of the code. Testing is an iterative process that involves feedback from the testing tool and the programmers capability to adjust to that feedback. Observations make this interaction more interesting and meaningful. The user can turn these observations into assertions which are checked on subsequent runs.

IV.3 Recommendation

In the end, there is no clear winner between the capabilities of Jtest and Agitator. If a decision needs to be made about which tool is better, one can only determine the better tool based on the needs of the software being developed and the set of tradeoffs associated with each tool.

For the case of Jtest, its main advantage is its test oracle. Every test is well documented in a generated JUnit code file. In todays corporate world where accountability is in high demand, these generated files function as a method for tracing test cases back to requirements. However, Jtest fails to generate sufficient test cases for float, double, and string types. It may be that these data types are not the most prevalent among modern software, but the gap in coverage remains. At the price of coverage, Jtest provides reasonable accountability.

Agitator has several advantages. It is able to obtain more complete coverage because it generates test data for float, double, and string types. It also goes beyond the call of duty of mere branch coverage, and generates extra random test inputs. It runs each method hundreds to thousands of times to test for robustness. Through its observations, it is capable of telling the programmer what the software is supposed to do. For all of Agitators coverage capability, its shortcoming may be its oracle facility. The user can only view a snapshot of one hundred test cases, and even this snapshot is not saved. Although this is understandable, because Ag-

itator generates thousands of tests which would take a sizeable chunk of memory to save, a programmer is not able to trace test cases back to requirements. In certain software engineering disciplines, it is required to correlate all requirements with tests. However, at the unit testing level, for which Agitator is intended, this traceability may not be important.

References

1999: *Code Conventions for the Java Programming Language*. Sun Microsystems.

Agitar, 2008: http://www.agitar.com.

Checkstyle4.4, 2008: http://checkstyle.sourceforge.net.

DeMillo, R., and Offutt, J., 1991: Constraint-based automatic test data generation. *IEEE Transactions on Computers*, **17**(9).

Eclipse, 2008: http://www.eclipse.org.

Ernst, M., Cockrell, J., Griswold, W., and Notkin, D., 2001: Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, **27**(2).

Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., and Xiao, C., 2007: The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, **69**(1–3).

Howden, W., 1975: Methodology for the generation of program test data. *IEEE Transactions on Computers*, **24**(5).

Howden, W., 1977: Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, **3**(4).

Howden, W., 1978: Algebraic program testing. *Acta Informatica*, **10**(1).

Javadoc, 2008: http://java.sun.com/j2se/javadoc/.

JUnit4.4, 2008: http://www.junit.org.

Korel, B., 1990: Automated software test data generation. *IEEE Transactions on Software Engineering*, **16**(8).

Offutt, J., Jin, Z., and Pan, J., 1999: The dynamic domain reduction approach to test data generation. *Software-Practice and Experience*, **29**(2).

Parasoft, 2008: http://www.parasoft.com.