1-1-2010

An Empirical Evaluation of the Effectiveness of JML Assertions as Test Oracles

Kavir Shrestha
University of Denver

An Empirical Evaluation of the Effectiveness of JML Assertions as Test Oracles

---

A Thesis

Presented to

the Faculty of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Kavir Shrestha

March 2010

Advisor: Dr. Matthew J Rutherford

Author: Kavir Shrestha
Title: An Empirical Evaluation of the Effectiveness of JML Assertions as Test Oracles
Advisor: Dr. Matthew J Rutherford
Degree Date: March 2010

ABSTRACT

Test oracles remain one of the least understood aspects of the modern testing process. An oracle is a mechanism used by software testers and software engineers for determining whether a test has passed or failed. One widely-supported approach to oracles is the use of runtime assertion checking during the testing activity. Method invariants, pre- and postconditions help detect bugs during runtime. While assertions are supported by virtually all programming environments, are used widely in practice, and are often assumed to be effective as test oracles, there are few empirical studies of their efficacy in this role. In this thesis, we present the results of an experiment we conducted to help understand this question. To do this, we studied seven of the core Java classes that had been annotated by others with assertions in the Java Modeling Language, used the muJava mutation analysis tool to create mutant implementations of these classes, exercised them with input-only (i.e., no oracle) test suites that achieve branch coverage, and used a machine learning tool, Weka, to determine which annotations were effective at "killing" these mutants. We also evaluate how effective the "null oracle" (in our case, the Java runtime system) is at catching these bugs. The results of our study are interesting, and help provide software engineers with insight into situations in which assertions can be relied upon to find bugs, and situations in which assertions may need to be augmented with other approaches to test oracles.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

While test oracles are a fundamental part of testing, they remain one of the least understood aspects of the testing process. Oracles help us determine whether a test has passed or failed. Several researchers have identified that assertions can be used as an oracle. However, not much work has been done in employing a runtime assertion checker as the test oracle engine and studying the effectiveness of assertions at detecting faults. This is the first extensive and rigorous work that has been done to study the effectiveness of the "implicit oracle" (which we refer to as the "null oracle" from here onwards) as well as assertions at detecting faults. In this thesis, we present the results of an experiment we conducted to evaluate the effectiveness of using Java Modeling Language (JML) assertions as test oracles. To do this, we created mutant versions of seven core Java classes that had been annotated by others and exercised them with input-only test cases written by us that achieved branch coverage. By doing this we were able to determine the effectiveness of both the "null oracle" (by turning off the annotations) and JML assertions at "killing" these mutants. To determine the effectiveness of JML assertions at catching bugs at different thresholds, we utilized Weka, a machine learning tool.

We worked with tools created by others in order to conduct our experiment. We used the common JML tools version 5.2 which contained annotated files for our seven test subjects. Working through the tool and learning the tool took a lot of effort. Although our subjects were annotated by others, we had to add certain annotations to all of them to get them to compile with the JML compiler (jmlc). A lot of other issues had to be dealt with as well which we describe in Section 4.1. For the purpose of this experiment more than 2050 mutants were generated and more than 800 were manually inspected to make sure they were valid. We also wrote randomized test cases for all of our test subjects to achieve branch coverage. The collection of method level metrics as described in Subsection 4.2.2 was also done manually. Overall, 28 data sets and four "Grand data sets" were created to determine the effectiveness of JML annotations at different thresholds.

The layout of this thesis is as follows: we start by introducing the major topics of our research below. In Chapter 2 we present an overview of the tools and techniques we used during our experiment viz. the null oracle, JML, jmlc, mutation testing, coverage criteria and Weka. We present related work in Chapter 3. Chapter 4 describes our experimental setup along with the metrics that we collected as well as our seven Java test subjects. In Chapter 5 we present a complete report of our results including the effectiveness of the "null oracle", effectiveness of JML assertions and classification of JML assertions at various thresholds of effectiveness. We end by presenting a discussion of the experiment, a summary of our major findings and by listing out our plans for future work in Chapter 6.

1.1 Software Engineering

Software engineering is the application of systematic, disciplined, quantifiable approaches to the development, operation and maintenance of software, and the study of these approaches [9]. One of the ten subdisciplines of software engineering is software testing, which is an activity performed for the purpose of improving the quality of a software system by identifying and eliminating defects and problems.

1.2 Software Testing

Software testing is the process of dynamic verification of the behavior of a program on a finite set of test cases which are selected from the usually infinite execution domain, against the expected behavior [9]. It is an indispensable step in the process of software development. While testing can only show the presence of failures and not their absence [1], it is still a very effective way of achieving software reliability. Software testing can also be stated as the process of verifying that a program meets the requirements and works as expected.

Testing can be categorized in different ways. If categorized using testing levels, then there are unit testing, integration testing and system testing; if categorized by objectives, there are acceptance/qualification testing, installation testing, alpha and beta testing, conformance testing, reliability testing, regression testing, performance testing, stress testing, back-to-back testing, recovery testing, configuration testing and usability testing; if categorized by code accessibility, there are whitebox/structural testing and blackbox/functional testing. Besides the above listed general categories, there are different ways to test special types of software. For example, there are different approaches to test object oriented software [62], [53].

The three main steps in testing are test case definition, test execution and test results evaluation. Test case definition is the basis of a successful test process. A successful test is one that finds an error. The more errors found by executing a set of test cases, the better those test cases are.

A test case is "A set of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test" [1].

Test cases are organized as one or more test sets/suites where a test set is "A set of test cases" [1], which is usually run as a group.

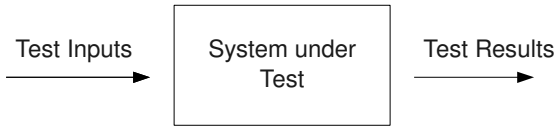

Figure 1.1: I-P-O Testing Model

Figure 1.1 [32], shows an Input-Process-Output (I-P-O) model for blackbox/functional testing. The test case here is a set of inputs and verification is done by observing the results. The test results are things such as program states for the system under test, software values, residual values left in memory, output data, etc.

All testing methods depend on the availability of an **oracle**, i.e., some mechanism that checks whether the system under test has behaved correctly or not in a particular execution. Test oracles are derived from the specifications of the system under test.

1.3 Oracles

An oracle is a mechanism used by software testers and software engineers for determining whether a test has passed or failed. We compare the output of the system

under test, for a given test case input, to the output that the oracle determines that a unit should have. Oracles are separate from the system under test. An ideal oracle is one that provides an unerring pass/fail judgment for any possible program execution, judged against a natural specification of intended behavior [5]. However, in the real world it is usually impossible to come up with an ideal oracle. As a result, based upon outputs, oracles are categorized as follows: true oracle, heuristic oracle, sampling oracle, consistent oracle and no oracle [33].

A "true oracle" reproduces all relevant results for a software under test (SUT) using independent platform, code, compiler, etc. In order to compare results, the same values are fed to the oracle and the system under test. A "heuristic oracle" reproduces only selected results for the system under test and the remaining values are checked using simpler algorithms based on a heuristic. A heuristic is applied to verify that the SUT returns values that are progressively larger or smaller than the last value. A "sampling oracle" uses a selected set of values where the values are selected based on some criteria besides statistical randomness. Boundary values, maximum, minimum, midpoints are examples that are often chosen. A "consistent oracle" is one where we verify current run results with the results of one test run which we specify as the oracle. The oracle here often comes from a simulator or an early version of the SUT. Lastly with "no oracle" we are only able to check that some results were produced and the program terminates normally. This represents a baseline oracle and is what we refer to as the "null oracle". We do not check the correctness of the results. Douglas Hoffman describes these different oracles in more detail [32].

Not only do formal specifications help in the derivation of test oracles, some view formal specifications as test oracles [3], [52]. Moving from the conventional way of implementing test oracles by comparing the test output with some pre-calculated

and presumably correct output, some formal specifications are able to monitor the specified behavior of the method under test to decide test success or failure. Also, the verification of an implementation can be checked by checking specification assertions during the execution of programs.

1.4 Assertions

Assertions are formal facts about the state of a program; they are typically implemented as a boolean expression that is true at certain points in program code [31]. Assertions are one of the most useful techniques for debugging, detecting faults and providing information about the internal state of a program. In order to determine the correctness of a program, a method or a segment of code, i.e. determine that it does what is intended, we need the ability to make assertions. If an assertion does not hold when the execution control reaches it, we know that either the program is wrong or the assertion is wrong. Even though assertions were initially developed in order to state the desired program behavior, it has found many other applications in software engineering like model checking which is a static approach to program verification [17].

Apart from their use in formal verification, assertions can also be viewed as a permanent defensive programming mechanism for runtime detection in production versions of software system [47]. This is where assertions have made their greatest impact, in the area of automated runtime fault detection, where formal assertion checks are instrumented into a program for execution along with the program's application logic [17]. Assertion features are supported by almost all programming languages. One popular approach is the use of macro statements that are expanded into appropriate program statements by preprocessors. The main examples are the assertion

facilities of C and C++ and their extensions [19], [45], [58], [65]. Here, assertions are boolean expressions that have been embedded into other program statements. If the boolean expressions do not hold at runtime, when the control reaches them, assertion failures are reported [16] and the program typically terminates abnormally. The Java programming language provides assertions as built-in statements after version 1.5. The assert statement was added to Java in version 1.4 but it didn't default on the compiler. It has to be explicitly turned on in 1.4 by specifying the "-source 1.4" option to the compiler. Since the built-in assertion is quite rudimentary, we do not use them but rather focus on the more fully-featured JML runtime assertion checker.

Chapter 2

Background

In this chapter we present the tools and techniques that are most important to the work described here. We start with a description of the null oracle in Section 2.1. We provide a brief overview of the Java Modeling Language (JML) in Section 2.2 along with an introduction to the JML compiler (jmlc) in Subsection 2.2.1. In Section 2.3 we introduce mutation analysis and the main ideas behind it. We conclude the chapter with an overview of coverage criteria (i.e., test adequacy criteria) in Section 2.4.

2.1 The Null Oracle

The "null oracle" is the implicit oracle that we have without doing any additional work. In our case the null oracle is the Java runtime system which helps us catch bugs in our code that cause the program to crash. If the program crashes (i.e., an exception is raised and propagated to the Java Virtual Machine) then the bug is caught as a result of having the null oracle and if the program terminates normally then the null oracle is not able to catch the bug in our code.

2.2 Java Modeling Language

The Java Modeling Language (JML) [14] is a behavioral interface specification language tailored to Java. It uses Hoare [31] style pre and postconditions and invariants that follow the design by contract (DBC) paradigm. JML blends Eiffel's design by contract approach [47] with the Larch tradition [28] and uses Java's expression syntax in assertions. JML assertions are either written in separate files or are written as special annotation comments to the Java program either after //@ or between /*@ .... @*/. The Java compiler simply ignores such assertions as comments. Consider the following example of a behavioral interface specification in JML, written as assertions in a Java program file, **'IntMathOps.java'**.

```
1  public class IntMathOps{
2    /*@ public normal_behavior
3      @   requires y >= 0;
4      @   assignable \nothing;
5      @   ensures 0 <= \result
6      @       && \result * \result <= y
7      @       && ((0 <= (\result + 1) * (\result + 1))
8      @           ==> y < (\result + 1) * (\result + 1));
9      @*/
10 public static int isqrt(int y)
11   {
12     return (int) Math.sqrt(y);
13   }
14 }
```

Listing 2.1: A Java file IntMathOps.java with assertions [40]

In Listing 2.1, the precondition is on line 3 following the keyword **requires**. The postcondition is on lines 5-8, following the keyword **ensures**. The precondition states what must be true about the arguments and when the precondition is true, the method must complete in a state that satisfies the postcondition. Hence, JML uses the **requires** clause to specify the caller's obligation and the **ensures** clause to specify the implementor's obligation [40]. The keyword **\result** is used in the postcondition to denote the actual value returned by the method. The type of **\result** is the return

type of the method. In the above example the postcondition states that the result is an integer approximation to the square root of the argument.

JML assertions can also be written in separate, non-Java files. This is mostly done when one does not control the source code, but wishes to annotate it. A filename with a suffix such as '.refines-java' (or '.refines-spec or '.refines-jml') would be used to add assertions to such a library or framework. The file with such a name would hold the assertions of the corresponding Java compilation unit. For example, if one wants to annotate the file IntMathOps2.java without modifying it then one would write assertions in the file 'IntMathOps2.refines-java', and include in that file the following refine-prefix [40].

```
refine ''IntMathOps2.java'';
```

In Listing 2.2 and Listing 2.3 we provide an example of how JML assertions can be kept in separate files. Listing 2.2 is the source file that does not contain any assertions and Listing 2.3 is the file that contains assertions. Since the files that contain assertions are not Java program files, JML requires the user to omit the code for concrete methods, as in a Java abstract method declaration. This is also depicted in Listing 2.3.

```
public class IntMathOps2                                    1
{                                                           2
    public static int isqrt(int y)                          3
    {                                                       4
        return (int) Math.sqrt(y);                          5
    }                                                       6
}                                                           7
```

Listing 2.2: IntMathOps2.java without assertions

```
//@ refine "IntMathOps2.java";

//@ model import org.jmlspecs.models.*;

public class IntMathOps2{
  /*@ public normal_behavior
    @   requires y >= 0;
    @   assignable \nothing;
    @   ensures 0 <= \result
    @       && \result * \result <= y
    @       && ((0 <= (\result + 1) * (\result + 1))
    @          ==> y < (\result + 1) * (\result + 1));
    @*/
public static int isqrt(int y);
}
```

Listing 2.3: IntMathOps2.refines-java with assertions [40]

JML allows declarations of various identifiers with the modifier **model**. One can declare model fields, methods and even types. A feature declared with **model** is only usable for specification purposes and is not available for use in Java code outside of assertions. Similarly, a model method is a method that can be called from assertions but cannot be called from ordinary Java code. A model field can be thought of as the abstraction of one or more non-model (i.e., Java or concrete) fields [40]. Model fields abstract values from concrete fields. In JML this is done with the help of a **represents** clause. Model methods and types, unlike model fields, are not abstraction of non-model methods or types. They are simply methods or types that help in specification.

A **ghost field**, similar to a model field, is also only used for the purposes of specification and cannot be used outside of assertions. However, unlike in model fields, **represents** clauses do not help determine the values of ghost fields. Their values are set directly during their initialization or with the use of set-statements embedded in the Java code.

In JML, class **invariants** are properties that have to hold for each reachable object in each publicly visible state, i.e., for each state outside of a public method

or constructor's execution and at the beginning and end of each public method's execution. It does not, however, have to hold during the execution of an object's methods.

In Listing 2.4 we provide an example of a specification that contains the declaration of a model field, an invariant and some method specifications.

```
  package org.jmlspecs.samples.stacks;                                  1
                                                                        2
//@ model import org.jmlspecs.models.*;                                3
                                                                        4
public abstract class UnboundedStack {                                  5
                                                                        6
/*@ public model JMLObjectSequence theStack;                           7
@ public initially theStack != null                                     8
@ && theStack.isEmpty();                                                9
@*/                                                                     10
                                                                        11
//@ public invariant theStack != null;                                 12
                                                                        13
/*@ public normal_behavior                                              14
@ requires !theStack.isEmpty();                                         15
@ assignable theStack;                                                  16
@ ensures theStack.equals(                                              17
@ \old(theStack.trailer()));                                            18
@*/                                                                     19
public abstract void pop( );                                            20
                                                                        21
/*@ public normal_behavior                                              22
@ assignable theStack;                                                  23
@ ensures theStack.equals(                                              24
@ \old(theStack.insertFront(x)));                                       25
@*/                                                                     26
public abstract void push(Object x);                                    27
                                                                        28
/*@ public normal_behavior                                              29
@ requires !theStack.isEmpty();                                         30
@ assignable \nothing;                                                  31
@ ensures \result == theStack.first();                                  32
@*/                                                                     33
public /*@ pure @*/ abstract Object top( );                             34
}                                                                       35
```

Listing 2.4: A file UnboundedStack.java with specifications [40]

In Listing 2.4, a model data field, *theStack*, is declared on the seventh line. At the end of the model field's declaration is an initially clause. Model fields cannot be explicitly initialized as there is no storage directly associated with them. However,

an initially clause can be used to describe an abstract initialization for a model field. The initially clause must be true of the field's starting value [40]. For our example, all reachable objects of the type UnboundedStack must have been created as empty stacks and subsequently modified using the type's methods.

Following the model field declaration on line 13 is a class invariant. In the example above, the invariant says that the value of *theStack* should never be null. Detailed description of model variables, invariants, initially and represents clauses and method specifications can be found in Leavens et al.'s JML Reference Manual [41].

2.2.1 JML Runtime Assertion Checker

The JML Runtime Assertion Checker (RAC) compiler (jmlc), like the Java compiler, produces Java bytecode from source code. It was developed at Iowa State University. It also adds assertion checking code to the bytecode that it produces. The bytecode produced by jmlc helps find inconsistencies between specifications and code by executing the assertions at runtime. It finds such inconsistencies dynamically, by executing JML's assertions while the program runs and notifies the user of any assertion violations [14]. The JML compiler not only helps us in the detection of software faults, but it helps check the correctness of JML assertions as well. Figure 2.1 depicts an overview of the JML environment. Further information about the JML Runtime Assertion Checker compiler can be found in Yoonsik Cheon's Technical Report [16].

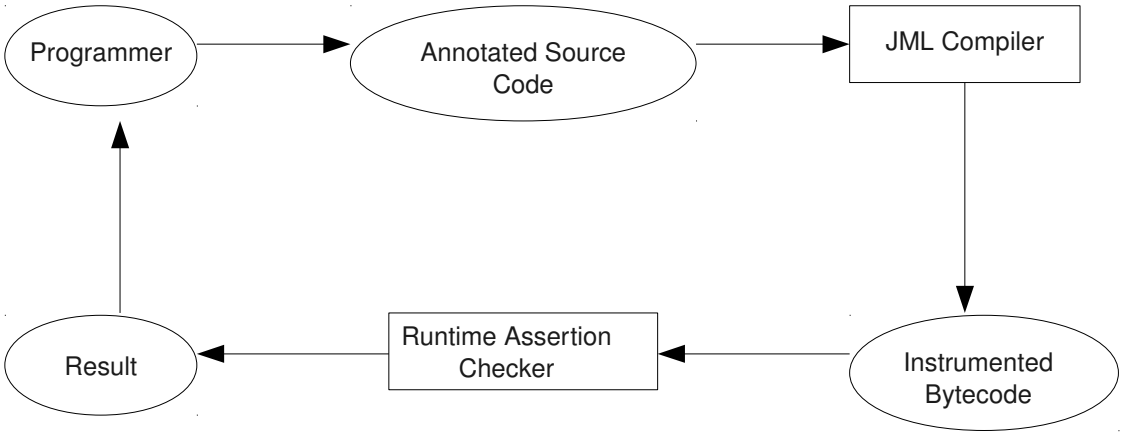

Figure 2.1: An overview of the JML environment [57]

We chose JML to test the effectiveness of assertions because it is an open source, fully-featured, runtime assertion system on a modern object-oriented language. Classes that have been annotated with JML are publicly available and the choice of Java meant that we were able to use other common tools like muJava, JUnit and Cobertura. A useful feature of JML is that it allows assertions to be kept separate from the source file. The fact that JML specifications are inherited by subclasses and classes implementing interfaces also suited our experiment.

2.3 Mutation Testing

Mutation testing is a method of software testing which involves modifying a program's souce code or byte code in small ways. The idea of mutation testing dates back to 1971, when Richard Lipton proposed the initial concepts of mutation in a class term paper entitled "Fault Diagnosis of Computer Programs" [50]. First research papers in this topic were published in the late '70s [12], [29], [21]. The DeMillo, Lipton and Sayward paper [12] is usually credited as the seminal reference.

During mutation testing, faults are introduced into a program by creating multiple

versions of the program, each of which contains one fault. Test data are then used to execute these faulty programs with the goal of causing each faulty program to produce different output than the original program. These faulty programs are mutants of the original and a mutant is considered to be killed when a test case causes it to produce output that is different from the original. The main concept behind mutation testing is that if we are able to come up with a test suite that kills all the valid mutants that have been generated, then that test suite will also be good at finding real bugs.

Mutations are always based on a set of "mutation operators" which specify syntactic variations of strings generated from a grammar. Mutation operators were derived from studies of programmer errors and correspond to simple errors that are typically made by programmers [37]. Mutation operators have been designed for various programming languages. Here are some that are listed by Offutt, Ammann and Liu [51] for, Fortran IV [60], COBOL [30], Fortran 77 [22], [37], C [20], Lisp [13], Ada [10], Java [35], and Java class relationships [43].

Mutation testing is done by selecting a set of mutation operators and applying them to the source program one at a time. Mutation testing is used to help the user measure and improve the quality of testing iteratively. Offutt, Lee, Rothermel, Untch and Zapf have done an experiment to determine sufficient mutation operators and have concluded that selective mutation is almost as strong as non-selective mutation [49]. A more detailed description of mutation analysis and mutation operators can be found in Ammann and Offutt's book Introduction to Software Testing [1]. The use of mutation analysis is common in empirical studies of software testing techniques, and is borne out by an empirical assessment conducted by Andrew, Briand and Labiche where they conclude that the use of mutation operators is an effective means of instrumenting faults when compared to hand seeded faults [2].

For our experiment, we used muJava, a mutation analysis tool, to create mutant

implementation of the seven Java classes. It automatically generates mutants for both traditional mutation testing and class-level mutation testing [44].

Listing 2.5 and Listing 2.6 provide an example of a mutant created for Listing 2.2 using muJava. We present both the original as well as the mutated version here.

```
1 public class IntMathOps2
2 {
3     public static int isqrt(int y)
4     {
5         return (int) Math.sqrt(y);
6     }
7 }
```

Listing 2.5: Original IntMathOps2.java

```
1 public class IntMathOps2
2 {
3     public static int isqrt(int y)
4     {
5         return (int) Math.sqrt(-y);
6     }
7 }
```

Listing 2.6: Mutated IntMathOps2.java

Listing 2.5 is the original Java class and Listing 2.6 is the mutated Java class produced by muJava after applying the Arithmetic Operator Insertion (AOI) mutation operator. This particular instance is a basic unary arithmetic operator insertion (AOI_U). The only difference between Listing 2.5 and Listing 2.6 is on line 5 which is depicted by the diff file in Listing 2.7 with an exclamation (!) mark.

```
      public static int isqrt(int y)
      {
!          return (int) Math.sqrt(y);
      }
}
       --- 3,7 ----

       public static int isqrt(int y)
       {
!          return (int) Math.sqrt(-y);
       }
}
```

Listing 2.7: Diff file representing the AOI mutation operator generated for IntMathOps2.java

2.4 Coverage Criteria

Coverage criteria are used to measure how well a program is exercised by a test suite, i.e., they help the engineer decide which test inputs to use. They are also a stopping condition which helps one decide when to stop testing. Effective use of coverage criteria makes it more likely that test engineers will find faults in a program, but satisfying a criterion does not guarantee any particular level of effectiveness at finding faults. Coverage criteria can sometimes be used to directly generate test case values to satisfy a given criterion [1].

Coverage criteria can be defined in terms of **test requirements**. Test requirements are specific items that must be covered or satisfied during testing. For statement coverage items consist of nodes whereas for branch coverage items consist of edges in the control flow graph. A coverage criterion is a set of rules and a process that define test requirements. Based on structure, we can divide coverage criteria into four categories [1]:

1. Graph based criteria

2. Logical Expressions criteria

3. Input Domain criteria

4. Syntactic Structures

Infeasible test requirements are test requirements that cannot be satisfied. If no test case values exist that meet the test requirements then they are labeled infeasible. For example, the presence of dead code results in infeasible test requirements as those statements can never be reached.

For our experiment we used graph coverage and in particular branch coverage. In order to achieve branch coverage, test cases need to execute every edge in the control flow graph of a program, i.e. all control transfers in the program under test are exercised. A more in-depth discussion about coverage criteria can be found in Amman and Offutt's book Introduction to Software Testing [1]. Zhu, Hall and May also do a good job of explaining various unit-testing coverage criteria as well as test adequacy [69].

Branch coverage was chosen as our coverage criterion as it takes a more in-depth view of the source code than simple statement coverage which just checks to see whether every node in a program has been executed. If all edges in a flow graph are covered, then all nodes are necessarily covered. In other words, branch coverage subsumes statement coverage. Branch coverage is also supported by many tools and is widely used in the industry.

To generate our coverage report, we chose Cobertura. Cobertura is a free Java tool that calculates the percentage of code executed by tests [1]. Cobertura is based on jcoverage, which is another code coverage utility for the Java platform. Cobertura instruments Java bytecode after it has been compiled and it can generate reports in HTML or XML. It gives us the percentage of lines and branches covered for each class,

[1] `http://cobertura.sourceforge.net/`

each package and the overall project. It also computes the McCabe cyclomatic code complexity of each class [46]. Figure 2.2 depicts a report generated by Cobertura for our subject NewHashSet. At the top of Figure 2.2, the overall statement and branch coverage values along with the cyclomatic code complexity of the class are shown. Along the left hand side, the number of times each statement has been executed is displayed.

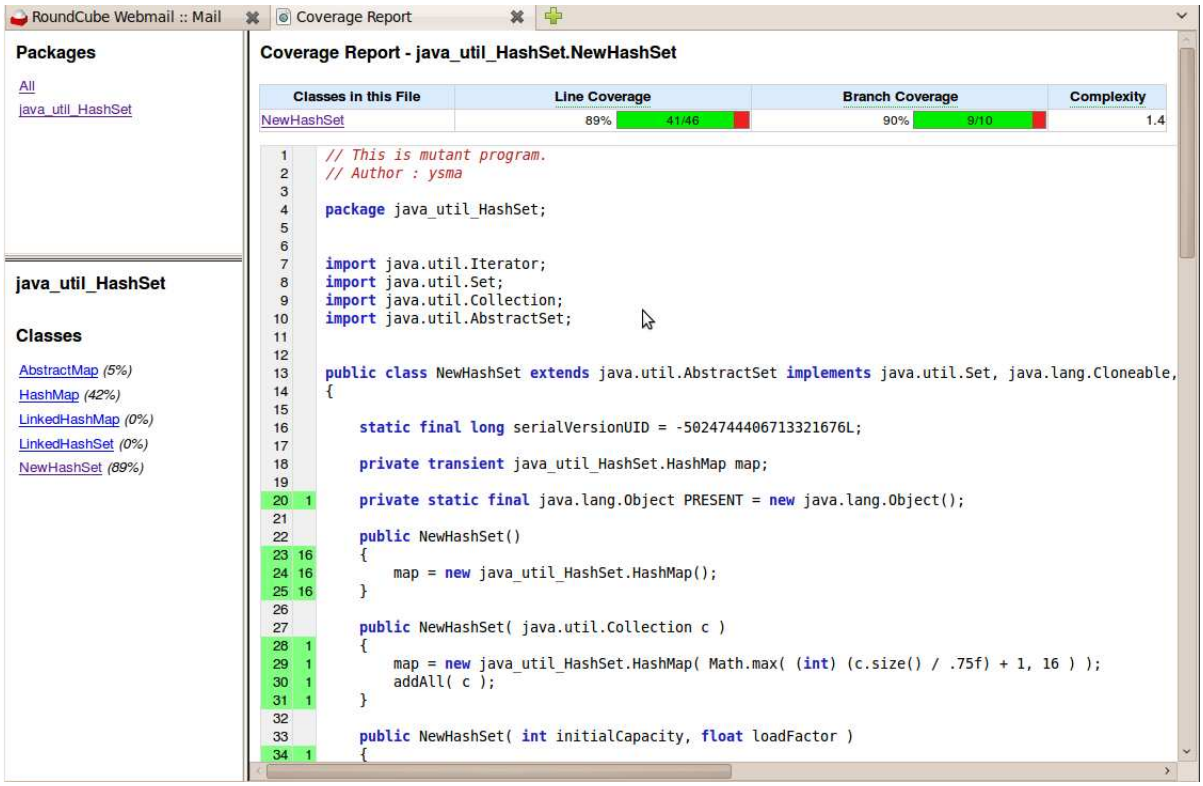

Figure 2.2: Report generated by Cobertura for NewHashSet

2.5 Weka

The Waikato Environment for Knowledge Analysis (Weka) is a comprehensive suite of Java class libraries that implement many state-of-the-art machine learning and

data mining algorithms [67]. Weka provides tools for pre-processing data, feeding the data into a variety of learning schemes and also analyzing the resulting classifiers and their performance. Weka was developed at the University of Waikato. The Weka workbench contains a collection of visualization tools and algorithms for data analysis and predictive modelling. Weka supports several standard data mining tasks such as data preprocessing, clustering, classification, regression, visualization and feature selection.

The primary learning methods in Weka are "classifiers" which induce a rule set or decision tree that models the data. Weka also includes algorithms for learning association rules and clustering data. Weka contains implementations of many algorithms for classification and numeric predictions such as ZeroR, OneR, NaiveBayes, DecisionTable, J48.J48, J48.PART, etc.

We use Weka's J48.J48 classification algoririthm to classify the effectiveness of JML assertions and come up with a decision tree. Weka's J48 algorithm is a Java implementation of the C4.5 algorithm [56] which generates a decision tree developed by Ross Quinlan. The decision trees generated by C4.5 are used for classification and hence C4.5 is often referred to as a statistical classifier. A more detailed description of Weka and the J48 algorithm can be found in Witten and Frank's book Data Mining: Practical Machine Learning Tools and Techniques [66].

In this work, we primarily used Weka as a black box to help us come up with decision trees that fit our empirical data at the different levels of effectiveness.

Chapter 3

Related Work

The roots of formal, logical assertions about program behavior came even before the existence of computers. The term "assertion" was used in Goldstine and vonNeumann's work on reasoning about programs to document invariants in algorithms [26]. Assertions were also used by Turing to document states that can be associated with various points in a routine [63].

The idea behind using assertions to help in software development is not new either. Floyd used loop assertions for program verification in 1967 [25]. Another work was by Hoare in 1969 where he used assertions to make formal statements about the behavior of programs [31]. Hoare famously defined "the proof schema {P} S {Q} (originally written by Hoare as P {S} Q), where S is a (composition of) statements, P is the precondition of S, and Q is the postcondition of S" [17]. Hoare interpreted his original schema P {S} Q as " If the assertion P is true before initiation of a program S, then the assertion Q will be true on its completion." After defining axiom schemas for primitive program statements and inference rules for compound statements and composition of statements, Hoare sketched a proof method whereby the programmer supplied a precondition P and postcondition Q for a program S and then applied the

system of proof rules to establish the validity of the statement {P} S {Q}. Luckham et al. elaborated Floyd's idea of using loop assertions into an algorithm for mechanical program verification which was based on the creation and proof of simple assertions called "verification conditions" [34].

In addition to helping in software development, assertions have been used to detect software faults during runtime. Several researchers have described systems that derive runtime consistency checks from simpler assertions [59], [8], [68]. Stucki and Foshee in their approach wrote assertions as annotations of FORTRAN source code which were translated by the preprocessor to embedded runtime checks that were invoked at appropriate times during the execution of the program [59]. Luckham and vonHenke developed the Anna annotation system to augment the Ada programming language [42]. Clarke and Rosenblum in their Impact Report, list seven extensive kinds of annotations that Anna provides [17]. These being subtype annotations, to specify a logical constraint on the set of values belonging to an Ada subtype; object annotations, to specify a logical constraint on the values a variable may hold; statement annotations, for specifying point assertions on the states following statement executions; subprogram annotations, to specify pre- and post conditions on subprograms; axiomatic annotations, to axiomatically or algebraically specify package behaviors; context annotations, to specify constraints on the items a compilation unit uses from the other compilation units it imports; and propagation annotations, to specify constraints on the way exceptions are propagated within a program.

The best known example of checking assertions at runtime is perhaps Bertrand Meyer's design-by-contract implemented in a programming language Eiffel [47]. Eiffel incorporated assertion constructs into the programming language, including support for pre and post conditions, initial values (in post conditions only), loop invariants, class invariants and a general assert statement. Eiffel was also used to specify behavior

at the design level. Not only is Eiffel suitable for runtime error checking but it also provides a good static analysis of semantic consistency. Eiffel's success in checking pre and post conditions contributed to the availability of similar facilities in other programming languages.

Other third party tools have also been designed that support design by contract. For Java, iContract by Reliable-Systems is one such tool [38]. iContract is a source code preprocessor which identifies annotated assertion expressions that use tags such as @pre to specify preconditions, @post to specify postconditions, etc. iContract converts these assertions into check code. iContract supports the use of class invariants, preconditions and postconditions. Additionally, it supports propagation of assertions via inheritance.

Besides the ones mentioned above there are numerous other runtime assertion checking facilities that have been developed. Here are some that are listed by Cheon [16] for programming languages, C++ [23], [27], [55], [65], .NET [4], Python [54], Smalltalk [15] and Java [6], [24], [36], [38], [18]. A comprehensive report on the history of runtime assertion checking can be found in Clarke and Rosenblum's IMPACT report [17].

Even though the use of runtime assertion checking is a widely-supported approach during the testing activity, there have only been a few empirical studies on their effectiveness at detecting faults. Rosenblum reports on several experiments that led to a classification of the assertions that were most effective at detecting faults in C programs [58]. Rosenblum describes a tool called APP, an Annotation PreProcessor for C. His work mainly strives to add annotations to existing C programs to describe the assertions. In his experiment Rosenblum carried out a case study on a C program subject. He found that the assertions that were written for the C program detected a high percentage of the discovered faults in the program. He also provided a clas-

sification of the assertions that were most effective at detecting faults so that future developers would reap the benefits of his efforts.

Muller et al. performed an experiment where they compared two assertion preprocessors, APP and Jcontract, in terms of their ability to aid software maintenance and extension tasks [48]. They found that programs produced with the help of assertions were more reliable and that assertions reduced the effort needed and made the effort more predictable.

Baudry et al. empirically validated the robustness and diagnosability factors of software using different measures by applying mutation analysis in a telecommunications switching system [7]. Baudry et al. defined robustness as "the degree to which software can recover for internal faults that would otherwise have provoked a failure" and diagnosability as the "degree to which the software allows easy and precise location of a fault when it is detected." They conclude that design by contract is a very efficient way of improving the diagnosability and robustness of a system and its general quality. They also state that the quality of contracts is more important than their quantity.

Vaos and Miller have done work to advocate the placement of assertions in code [64]. Excessive assertion uses slows down the execution speed and there might not be any cost-benefits between the performance degradation and the potential benefits of finding faults. They claim that assertions should be placed in locations where traditional testing is unlikely to uncover software faults and advocate the use of sensitivity analysis. Sensitivity analysis makes predictions concerning future program behavior by estimating the effect that input distribution, syntactic mutants and changed values in data states have on current program behavior [64].

According to Clarke and Rosenblum [17] the most definitive study to date regarding the impact of assertions on faults in production software systems is the study

undertaken by Microsoft researchers [39]. Their study revealed that there existed a "statistically negative correlation between the assertion density and fault density." Code that had been developed and tested with assertions had fewer faults than code that did not and as the density of assertions in the code increased, the number of faults decreased.

Several researchers have identified that specification can be used as an oracle. However, not much work has been done in employing a runtime assertion checker as the test oracle engine. Yoonsik Cheon and Gary T. Leavens use a runtime assertion checker as the decision procedure for test oracles [16]. They monitor the specified behavior of the method under test to decide test failure or success. Peters and Parnas have also generated test oracles from formal program specifications [52]. A relational program specification is used to specify the behavior of a program and the test oracle procedure generated in C and C++ checks if an input and output pair satisfies the relation described by the specification. Their approach tests for only pre and post conditions and the specification by themselves aren't used as test oracles.

Another approach is by Antoy and Hamlet who check the execution of an abstract data type's implementation against its specification [3]. They use an algebraic specification which serves as a test oracle. The algebraic specification is executed by rewriting and compared with the implementation's execution. In order to compare, the user provides an abstraction function which maps implementation status to abstract values.

Briand, Labiche and Sun have investigated how the instrumentation of contracts addresses the definition and coding of test oracles as well as the isolation of faults once failures have been detected [11]. They investigate in detail the impact of contract executable assertions on diagnosability. They perform a thorough case study where they define OCL (Object Constraint Language) contracts, instrument them using a

commercial tool (JContract) and assess the benefits and limitations of contracts by comparing the results of programs instrumented with contracts and programs that exclusively use test oracles. They present an ATM case study and generate mutant versions to compute diagnosability for program versions using only test oracles and instrumented versions using contracts. They discover that the average diagnosability for the ATM program when using contract is at least eight times better than using just test oracles and that hence it leads to significant effort savings.

Tan and Edwards performed a study to evaluate the effectiveness of JML-JUnit Testing [61]. They test the JML-JUnit unit testing strategy using mutation analysis. They chose six classes from the java.util library and chose Jester [1], a mutation testing tool to generate mutants for the classes. After generating mutants they ran the JUnit test cases by providing preselected values for scalars and provided null and the default constructor for object types. Their results show that the JML-JUnit testing strategy is not very effective but they were able to detect bugs in the specifications of classes. However, their study raises some questions. They used Jester to generate bugs which does not provide a comprehensive set of mutations.

Our contributions are most closely related to Tan and Edwards' study [61]. We both use mutation analysis to evaluate the effectiveness of testing with assertions. However, rather than just generating mutants and seeing how many mutants were caught and how many weren't, we analyze the effectiveness of different annotations at different thresholds. We also evaluate how effective the Java runtime system is when it comes to finding bugs. They used Jester to create faulty implementations of the Java classes where as we used muJava. Jester does not even cover all of the traditional mutation operators. Also, Tan and Edwards' only experimented with six classes from the java.util library where as we test with seven, one of which stems from

[1] `http://jester.sourceforge.net`

the java.net library. Unlike Tan and Edwards we also break down the classes in order to evaluate the effectiveness of JML annotations at the method-level.

Chapter 4

Experimental Method

In this chapter we describe the experimental setup, the metrics that we collected, and the seven Java classes that are the subjects of our experiment. Section 4.1 lists the steps that we took when conducting our experiment. Section 4.2 provides the set of class as well as method-level metrics that we collected and in Section 4.3 we describe the seven core Java classes which serve as subjects in our experiment.

4.1 Experimental Infrastructure

We downloaded the common JML tools version 5.2 (JML.5.2) [1]. Since JML only works with J2SDK 1.4.2 and earlier versions, we also downloaded J2SDK 1.4.2 and got that set up. We also downloaded and set up muJava - a Mutation Analysis Tool and Cobertura - a Java Coverage Tool. muJava was needed to create mutant versions of our subjects and Cobertura to generate our coverage reports. Once we had the tools that were needed, the next step was to get the source code for the different Java libraries as those were going to be our subjects. We obtained the Java 1.4 source files

[1] `http://sourceforge.net/projects/jmlspecs/`

from the Sun Microsystems website [2].

Once all the tools were set up, the different Java library classes were compiled with the JML compiler (jmlc). We had to rename the JML specification files which came with the JML tools to *.refines-java and also add a //@ refine "*.java" statement to the refines-java file. To get the classes to compile with the JML complier certain /*@ nullable @*/ and /*@ non_null @*/ assertions had to be added to various methods of the refines-java file as well as the source file itself. With certain classes, some ghost field errors had to be fixed and in some cases there were missing represents clauses that had to be included to represent the declared model variables.

After all of the JML errors were taken care of, test cases were written for the different Java classes. The goal was to achieve 90% branch coverage. In order to generate our coverage reports we used Cobertura. To generate mutant versions of our test subjects, we used muJava. Traditional method-level mutation operators were used to create the mutants.

Once we had our mutants and our test cases, we ran the test cases to inspect how many mutants were killed and how many mutants were still alive. When running the test cases on the mutants, the first run did not include assertions (referred to below as the "null oracle") and the second run did include the JML assertions. All the mutants that were not caught by the null oracle were then classified to see whether they were valid or invalid. Invalid mutants are mutants that represent infeasible test requirements as they always produce the same output as the original program. An example of invalid mutants are equivalent mutants which are functionally equivalent to the original program. Replacing a "==" with a "<=" would result in an equivalent mutant. Of the remaining valid mutants that were not killed by the null oracle during the first run we checked to see how many of them that contained assertions were killed

[2] `http://www.sun.com/`

during the second run as a result of using JML assertions as a test oracle. In this way, we were able to make a comparison between the number of mutants killed by the null oracle itself and the mutants which were not killed by the null oracle and had assertions that were killed as a result of JML assertions.

Even though we had the results for the effectiveness of the null oracle at killing mutants as well as the effectiveness of JML assertions, used as the test oracle, at killing mutants we still did not have enough information to evaluate the effectiveness of JML assertions at different thresholds. The thresholds that we used for our experiment were 70%, 80%, 90% and 100%. In order to do that we collected various metrics as listed in Section 4.2. We had to manually collect all of the method-level metrics whereas as Cobertura helped generate the class metrics. Once we had collected all the metrics, we created a data set in the Attriture-Relation File Format (ARFF) for each of our test subjects, for thresholds of effectiveness ranging from 70% to 100%. Overall, we created 28 data sets for our seven test subjects. After creating the 28 data sets, we merged the data sets for each threshold of effectiveness together thus giving us four "Grand Data Sets." We fed these four data sets to Weka and utilized the J48 classification algorithm to come up with a decision tree for each of the four thresholds of effectiveness. We present these results in Section 5.4.

4.1.1 A Sample Run Through the Experiment

Below is an example of an entire run-through of our experimental setup with the Java utility class HashSet.

Step 1 - We created a package java.util and hence had to rename HashSet as NewHashSet as we wanted the Java and jml compiler to use our HashSet (NewHashSet.java) and not the original java.util.HashSet. At first we tried

creating a package java_util_HashSet and using HashSet from the package but this was causing class loader issues.

Step 2 - We copied over the specification file for HashSet - HashSet.refines-spec into our package under java.util and renamed it to NewHashSet.refines-java. We also added a //@ refine "NewHashSet.java" clause. Since NewHashSet.refines-java refines the NewHashSet.java file, we had to add 'also' to all of the method specifications in the NewHashSet.refines-java file.

Step 3 - To get both of the files to compile with the jml compiler, we had to add /*@ nullable @*/ and /*@ non_null @*/ assertions at certain places. Listing 4.1 shows an example of us adding the /*@ non_null @*/ assertion to the **clone()** method in NewHashSet.java and Listing 4.2 shows an example of us adding the /*@ non_null @*/ and 'also' assertions to NewHashSet.refine-java.

```
public /*@ non_null @*/ java.lang.Object clone()                    1
{                                                                   2
    try {                                                           3
          java.util.NewHashSet newSet =                             4
          (java.util.NewHashSet) super.clone();                     5
          newSet.map = (java.util.HashMap)                          6
          map.clone();                                              7
          return newSet;                                            8
    } catch ( java.lang.CloneNotSupportedException e ) {            9
          throw new java.lang.InternalError();                      10
    }                                                               11
}                                                                   12
```

Listing 4.1: /*@ non_null @*/ assertion added to NewHashSet.java

```
/*@ also                                                                    1
  @   public normal_behavior                                               2
  @     assignable \nothing;                                               3
  @     ensures \result instanceof Set && \fresh(\result)                  4
  @          && ((Set)\result).equals(this);                               5
  @     ensures_redundantly \result != this;                               6
  @*/                                                                      7
public /*@ non_null */ Object clone();                                     8
```

Listing 4.2: also assertion added to NewHashSet.refines-java

Step 4 - Besides the above assertions we also had to add certain **represents** and **in** clauses to the original HashSet.refines-spec file. The represents and in clauses caused us a lot of trouble at first. Certain classes that we looked into already had model variables being represented with represents clauses whereas for other classes although the model variables were defined and being used throughout the class, they were never actually being represented. In Listing 4.3 we present the original HashSet.refines-spec file with all of their defined model variables, invariants and represents clauses and in Listing 4.4 we present our NewHashSet.refines-java file with all of their assertions as well as our added in and represents clauses.

```
public class HashSet extends AbstractSet                                  1
    implements Set, Cloneable, java.io.Serializable                        2
{                                                                          3
    //@ public represents addOperationSupported = true;                    4
    //@ public represents removeOperationSupported = true;                 5
                                                                           6
    //@ public model int initialCapacity;                                  7
    //@ public model float loadFactor;                                     8
                                                                           9
    //@ public invariant initialCapacity >= 0;                             10
    //@ public invariant loadFactor > 0;                                   11
                                                                           12
    /*@ public normal_behavior                                             13
      @   assignable theSet, initialCapacity, loadFactor;                  14
      @   ensures theSet != null && theSet.isEmpty();                      15
      @   ensures loadFactor == 0.75;                                      16
      @   ensures initialCapacity == 16;                                   17
      @*/                                                                  18
   public HashSet();                                                       19
```

Listing 4.3: Original HashSet.refines-spec file provided by JML

```
public class NewHashSet extends AbstractSet                                    1
    implements Set, Cloneable, java.io.Serializable                             2
{                                                                               3
    //@ public represents addOperationSupported <- true;                       4
    //@ public represents removeOperationSupported <- true;                    5
    //@ public represents nullElementsSupported <- true;                       6
                                                                                7
    //@ public model int initialCapacity;                                      8
    //@ public model float loadFactor;                                         9
                                                                                10
    private /*@ spec_public @*/ final int _initialCapacity;                     11
    //@ in initialCapacity;                                                     12
    private /*@ spec_public @*/ final float _loadFactor;                        13
    //@ in loadFactor;                                                          14
    private /*@ spec_public @*/ transient HashMap map;                          15
    //@ in theSet;                                                              16
                                                                                17
    //@ public represents initialCapacity <- _initialCapacity;                 18
    //@ public represents loadFactor <- _loadFactor;                           19
                                                                                20
    //@ public represents theSet <-                                            21
    //@ JMLEqualsSet.convertFrom(map.keySet());                                 22
                                                                                23
    //@ public invariant initialCapacity >= 0;                                 24
    //@ public invariant loadFactor > 0;                                       25
                                                                                26
    /*@ also public * normal_behavior                                          27
      @ assignable theSet, initialCapacity, loadFactor;                         28
      @ ensures theSet != null && theSet.isEmpty();                             29
      @ ensures loadFactor == 0.75;                                             30
      @ ensures initialCapacity == 16;                                          31
      @*/                                                                       32
   public NewHashSet();                                                         33
```

Listing 4.4: Altered NewHashSet.refines-java file with additional in and represents clauses

Step 5 - The next step was to write test cases for the class NewHashSet.java. NewHashSet was a fairly simple class hence we did not face too many issues writing test cases for it. For some other complex classes like Date and URL this process was quite challenging. The **parse(String)** method for Date was one that was really challenging. After writing the test cases we generated a coverge report using Cobertura. For NewHashSet we were able to achieve 90% branch coverage with 20 test cases. The process of generating Cobertura coverage reports was quite troublesome as well. Since we had our package named as

java.util and java.net in order to eliminate class loader issues in Step 1, we were now getting security exceptions that told us that we were using a prohibited package name: java.util. We basically had to create two packages for every test subject one named java.util to get the jml compiler to work and the other java_util_SubjectName for us to be able to generate Cobertura reports. The files in these packages were identical except for the 'package' statement.

Step 6 - Next we used muJava to generate mutant versions of NewHashSet.java. This step was not as troublesome as some of the others but for certain classes we had to comment out assert statements as those was causing Open muJava exceptions. Listing 4.5 shows us an example of a diff file that represents one of the mutants that muJava generated.

```
 public boolean remove( java.lang.Object o )                    1
{                                                               2
!       return map.remove( o ) == PRESENT;                      3
 }                                                              4
                                                                5
 public void clear()                                            6
  --- 86,92 ----                                               7
                                                                8
public boolean remove( java.lang.Object o )                     9
{                                                               10
!       return map.remove( o ) != PRESENT;                      11
}                                                               12
                                                                13
 public void clear()                                            14
```

Listing 4.5: Diff file representing the Relational Operator Replacement mutation opearator generated for NewHashSet.java

Step 7 - Next we ran the test cases on the mutants. During the first run we did not include the JML assertions. In the second run we did. We then classified all the mutants that were not caught by the null oracle to see if they were valid or invalid. Of the remaining valid mutants that were not killed by the null oracle during the first run, we checked to see how many of them that contained

assertions were killed during the second run.

The results that we got for NewHashSet are listed in Table 4.1 which reflect our results in Sections 5.1 and 5.2. A method-level breakdown of the results for NewHashSet is presented in Subsection 5.3.2.

Total number of mutants generated by muJava	99
Number of invalid mutants	6
Total number of valid mutants	93
Mutants not caught by the null oracle during the 1^{st} run	76
The total number of mutants not caught by the null oracle with assertions	42
The number of mutants caught by JML assertions during the 2^{nd} run	24 (57.14%)

Table 4.1: Results for NewHashSet

Step 8 - Once we had our results as described in Step 7, we were able to determine the effectiveness of the null oracle and JML assertions used as the test oracle at catching bugs. In order for us to evaluate the effectiveness of JML assertions at different thresholds, we collected the metrics as listed in Section 4.2 for NewHashSet. The class level metrics were generated by Cobertura but we collected all of the method-level metrics manually. After collecting all the metrics, we created a data set in the ARFF file format for us to be able to utilize Weka's J48 algorithm to help us come up with a decision tree to classify the JML assertions at various thresholds of effectiveness. Figure 4.1 depicts a data set that was created for NewHashSet at 70% effectiveness.

```
@relation NewHashSet_70

@attribute statements real
@attribute branches real
@attribute preothers real
@attribute postothers {yes,no}
@attribute postself real
@attribute preself real
@attribute classinvariants real
@attribute testcases real
@attribute modelvariables real
@attribute modelvariablesused real
@attribute modelmethods real
@attribute modelmethodsused real
@attribute 70percentmutantscaught {yes, no}

@data
3,1,0,no,4,0,2,1,3,4,0,0,yes
3,1,0,no,4,1,2,1,3,4,0,0,yes
4,1,0,no,2,1,2,1,3,2,0,0,no
3,1,0,no,0,0,2,0,3,0,0,0,no
1,1,0,yes,0,0,2,2,3,2,0,0,yes
1,1,0,yes,0,0,2,2,3,1,0,0,no
6,2,0,no,0,0,2,2,3,0,0,0,no
```

Figure 4.1: Weka Data Set for NewHashSet at 70% Effectiveness

The attributes that we use in our data sets as shown in Figure 4.1 are number of statements, number of branches, number of preconditions on others, presence of postcondition on others, number of postconditions on self, number of preconditions on self, number of class invariants, number of explicit test cases, number of model variables defined, number of model variables used, number of model methods defined and the number of model methods used. The final attribute is the "outcome" which tells us whether the threshold for effectiveness was reached or not (for this specific example whether 70% of the valid mutants that were not caught by the null oracle, that were annotated, were caught as a result of using JML assertions as the test oracle). We converted the final attribute from a numerical value as represented in the JML Caught Weighted % column in Section 5.3 to a boolean value as we just wanted to see whether 70% of the bugs were caught or not.

4.2 Metrics

In order to help us classify the effectiveness of JML assertions we collected some metrics for all of our test subjects listed in Section 4.3. We provide a list as well as a brief introduction of the various metrics collected in this section. Subsection 4.2.1 describes class metrics and Subsection 4.2.2 describes method-level metrics.

4.2.1 Class Metrics

The class metrics we collected are source lines of code (executable lines of code), number of methods for each class, number of branches in each class, the number and percentage of lines our test suite covered for each class (statement coverage numbers), the number and percentage of branches our test suite covered for each class (branch coverage numbers), the McCabe's cyclomatic code complexity [46] and the number of test cases written for each class. All of the metrics besides the number of test cases listed above are generated by Cobertura.

4.2.2 Method-level Metrics

The method-level metrics we collected with their abbreviations provided in parenthesis () to fit them all in a single table are as follows: number of executable statements in the method (statements), number of branches in the method (branches), number of preconditions on other methods that affect this method (Pre O), number of postconditions on other methods that affect this method (Post O), number of preconditions for the method itself (Pre S), number of postconditions on the method itself (Post S), explicit number of test cases to test that method (Explicit Test Cases) and number of valid mutants that muJava generated for that method (Mutants). All of the metrics mentioned above were manually collected.

There are a few metrics listed above that need further clarification. By preconditions on other methods that affect this method we mean preconditions on some other (caller) method that calls the method under test (callee) and by postconditions on other methods that affect this method we mean the postconditions on some other (caller) method that calls the method under test (callee) as shown in Figure 4.2.

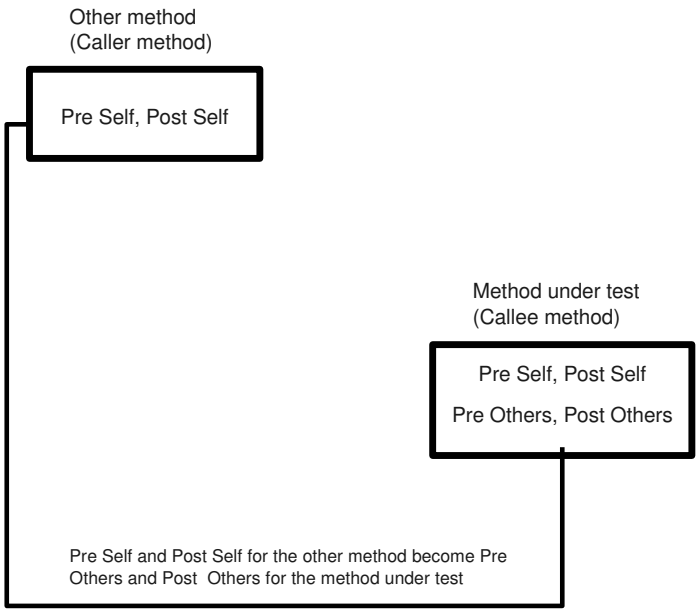

Figure 4.2: Pre and Postconditions on Others

Now let us take a look at a specific example in Date. Date has two methods **getTime()** and **getTimeImpl()**. The method getTime() simply calls the method getTimeImpl(). Here getTime() is the caller and getTimeImpl() is the callee. The method getTimeImpl() has no assertions whereas getTime() is annotated (we tested subjects with assertions written by others). muJava generated 6 mutants for getTimeImpl() out of which 2 were caught by the null oracle. Even though getTimeImpl() didn't have any assertions, the remaining 4 mutants were still killed during the 2nd run as shown in Table 5.3. This is where the preconditions of others and postcondi-

tions of others come into play. The caller method getTime() has 2 preconditions and 2 postconditions which helped kill the remaining 4 mutants for getTimeImpl() (callee). The 2 preconditions and 2 postconditions of getTime() serve as the 2 precondition on others (Pre O) and the 2 postcondition on others (Post O) for getTimeImpl() as shown in Table 4.3.

4.3 Subjects

As our subjects, we chose seven Java classes. Of the seven subjects, six were from the Java Utility package: Date, HashSet, Observable, Stack, TreeSet and Vector. One was from the Java Net package: URL. Below we provide a brief description of these classes along with the class metrics and the coverage numbers we are able to get which are generated by Cobertura. After the class metrics and coverage numbers, we also provide a method-level metrics that we have collected for each class.

4.3.1 Date

Date represents a specific time, with millisecond precision [3]. Listed below are the metrics for Date along with our coverage numbers.

Source Lines of Code	263
Number of methods	40
Number of branches	204
Number of lines covered	218 (82%)
Number of branches covered	168 (82%)
Cyclomatic complexity	3.6
Number of test cases	64

Table 4.2: Metrics for Date

For Date, we were able to get a pretty high number for branch coverage of 82%.

[3] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/Date.html`

Most of the complexity in Date was in the method long parse() which did not have any assertions. In Table 4.3, we provide a breakdown of the class Date based on its methods.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
boolean after(Date)	1	1	0	0	4	2	2	6
boolean before(Date)	1	1	0	0	4	2	2	1
boolean equals(Date)	1	2	0	0	1	2	3	9
Date(int,int,int)	1	1	0	0	0	0	1	12
Date(int,int,int,int,int)	1	1	0	0	0	0	1	20
Date(int,int,int,int,int,int)	6	4	0	0	0	0	1	32
Date(long)	2	1	0	0	0	2	1	4
int compareTo(Date)	3	4	2	1	6	3	4	24
void setTime(long)	2	2	0	0	2	2	2	8
long getTimeImpl()	1	2	2	2	0	0	2	6

Table 4.3: Method-level metrics for Date

4.3.2 HashSet

HashSet implements the Set interface, backed by a hash table. It makes no guarantees as to the iteration order of the set but it does guarantee that the order will remain constant over time [4]. Listed below are the metrics for HashSet along with our coverage numbers.

Source Lines of Code	46
Number of methods	15
Number of branches	10
Number of lines covered	41 (89%)
Number of branches covered	9 (90%)
Cyclomatic complexity	1.4
Number of test cases	20

Table 4.4: Metrics for HashSet

For HashSet, we were able to get a high number for branch coverage as well of 90%. HashSet was a simpler class in terms of complexity as it only had 10 branches.

[4] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/HashSet.html`

In Table 4.5, we provide a breakdown of HashSet based on its methods.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
HashSet(int)	3	1	0	0	0	4	1	10
HashSet(int,float)	3	1	0	0	1	4	1	18
HashSet(Collection)	4	1	0	0	1	2	1	16
HashSet(int,float,boolean)	3	1	0	0	0	0	0	22
boolean add(Object)	1	1	0	2	0	0	2	2
boolean remove(Object)	1	1	0	1	0	0	2	2
void readObject(Object)	6	2	0	0	0	0	2	23

Table 4.5: Method-level metrics for HashSet

4.3.3 Observable

Observable represents an observable object or data in the model-view paradigm. An observable object can have one or more observers. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by the call to their update method [5]. Listed below are the metrics for Observable along with our coverage numbers.

Source Lines of Code	30
Number of methods	10
Number of branches	8
Number of lines covered	19 (63%)
Number of branches covered	4 (50%)
Cyclomatic complexity	1.6
Number of test cases	8

Table 4.6: Metrics for Observable

We were only able to achieve 50% branch coverage for Observable as the test cases which exercised the remaining branches caused JMLEntryPreCondition Errors. As a result those test cases had to be turned off. An entry precondition error refers to an assertion violation concerned with the precondition of the method. In Table 4.7, we

[5] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/Observable.html`

provide a breakdown of Observable based on its methods. The * in the Explicit Test Cases column below implies implicit test cases. What we mean by those is that we did not write an explicit test case to test that method but to test other methods we called that method, hence it was tested implicitly. For example, if we were testing the remove method for a List, then for certain test cases we would first add an item to the List and then remove it. In this case add is being tested implicitly by the test case for remove.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
void notifyObservers(Object)	8	4	1	0	1	0	1*	17
void addObserver(Observer)	4	4	0	0	1	1	2	3
boolean hasChanged()	1	1	0	0	0	1	1	1

Table 4.7: Method-level metrics for Observable

4.3.4 Stack

Stack represents a last-in-first-out (LIFO) stack of objects. It provides the usual push and pop operations as well as a method to peek at the top item on the stack [6]. Listed below are the metrics for Stack along with our coverage numbers.

Source Lines of Code	17
Number of methods	6
Number of branches	6
Number of lines covered	17 (100%)
Number of branches covered	6 (100%)
Cyclomatic complexity	1.67
Number of test cases	10

Table 4.8: Metrics for Stack

Since Stack was a small class with only 6 branches it was not hard to achieve 100% branch coverage. In Table 4.9, we provide a breakdown of Stack based on its methods.

[6] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/Stack.html`

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
boolean empty()	1	1	0	0	0	1	2	0
int search(Object)	4	2	0	0	2	3	2	14
Object peek()	4	2	0	0	1	2	2	18
Object pop()	5	1	0	0	1	4	2	8

Table 4.9: Method-level metrics for Stack

4.3.5 TreeSet

TreeSet implements the Set interface backed by a TreeMap instance which guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements [7]. Listed below are the metrics for TreeSet along with our coverage numbers.

Source Lines of Code	59
Number of methods	22
Number of branches	22
Number of lines covered	42 (71%)
Number of branches covered	11 (50%)
Cyclomatic complexity	1.55
Number of test cases	19

Table 4.10: Metrics for TreeSet

For TreeSet too we were only able to achieve 50% branch coverage as the test cases which exercised the remaining branches caused JMLInternalNormalPostcondition Errors. As a result those test cases had to be turned off. Overall, for TreeSet we had to turn off 5 test cases. An internal normal postcondition error notifies internal normal postcondition violations. In Table 4.11, we provide a breakdown of TreeSet

[7] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/TreeSet.html`

based on its methods.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
boolean add(Object)	1	1	2	5	0	0	1	2
boolean addAll(Collect	9	4	4	1	0	0	3	31
boolean remove(Object)	1	1	2	6	0	0	1	2
void readObject(Object)	6	1	0	0	0	0	2	7

Table 4.11: Method-level metrics for TreeSet

4.3.6 URL

URL represents a Uniform Resource Locator (URL), a pointer to a resource on the World Wide Web. A resource can be something as simple as a file or a directory or other complicated objects such as a query to a database [8]. Listed below are the metrics for URL along with our coverage numbers.

Source Lines of Code	244
Number of methods	33
Number of branches	156
Number of lines covered	152 (62%)
Number of branches covered	81 (51%)
Cyclomatic complexity	3.91
Number of test cases	29

Table 4.12: Metrics for URL

For URL, we were able to achieve 51% branch coverage. URL was a complex class with complicated private methods. Also we needed to turn off some of our test cases for URL as they were causing some JML errors. In Table 4.13, we provide a breakdown of URL based on its methods. The * in the Explicit Test Cases column

[8] `http://sun.java.com/j2se/1.4.2/docs/api/java/net/URL.html`

below implies implicit test cases.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
boolean isValidProtocol(String)	11	8	0	0	0	0	6	68
int getPort()	1	1	0	0	0	1	1	4
URL(String,String,int,String)	1	1	0	8	0	1	5	4
URL(String,String,int,String,..)	25	14	0	0	0	8	5*	56
URL(String,String,String)	1	1	0	9	0	1	1	1
void set(String,String,int,..)	11	1	0	0	0	9	1	7

Table 4.13: Method-level metrics for URL

4.3.7 Vector

Vector implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as need to accommodate adding and removing items [9]. Listed below are the metrics for Vector along with our coverage numbers.

Source Lines of Code	192
Number of methods	48
Number of branches	84
Number of lines covered	183 (95%)
Number of branches covered	80 (95%)
Cyclomatic complexity	2.24
Number of test cases	73

Table 4.14: Metrics for Vector

We were able to achieve 95% branch coverage for Vector. In order to achieve 95% branch coverage we wrote 73 test cases. In Table 4.15, we provide a breakdown of Vector based on its methods. The * in the Explicit Test Cases column implies implicit

[9] `http://sun.java.com/j2se/1.4.2/docs/api/java/util/Vector.html`

test cases.

Method Name	statements	branches	Pre O	Post O	Pre S	Post S	Explicit Test Cases	Mutants
boolean add(Object)	3	1	0	0	1	1	1*	4
boolean addAll(int,Collection)	10	3	6	1	0	0	4	12
boolean addAll(Collection)	7	1	4	1	0	0	2	4
boolean removeElement(Object)	5	2	0	0	2	3	2	0
int indexOf(Object,int)	7	10	0	0	3	4	5*	2
int lastIndexOf(Object)	1	1	3	4	2	3	5	4
int lastIndexOf(Object,int)	8	8	0	0	4	4	5*	2
Object lastElement()	3	2	0	0	2	1	2	4
Object remove(int)	7	4	0	0	3	1	4	12
Vector(Collection)	3	1	0	0	1	1	1	8
void addElement(Object)	3	1	0	0	14	8	1	4
void ensureCapacity(int)	2	1	0	0	2	2	1	0
void ensureCapacityHelper(int)	8	4	0	0	0	0	0	8
void insertElementAt(Object,int)	7	2	0	0	16	5	3*	13
void removeAllElements()	4	2	0	0	0	1	2*	1
void removeElementAt(int)	11	6	0	0	4	1	4	13
void removeRange(int,int)	6	2	6	1	0	0	1	12
void setSize(int)	7	4	0	0	4	3	2	1
void trimToSize()	6	2	0	0	0	1	2	0

Table 4.15: Method-level metrics for Vector

Apart from the seven subjects listed above we also looked into a few more Java libraries which we couldn't carry on with because of various problems faced. We had class loader issues with certain classes and for others we were able to compile the classes with the JML compiler (jmlc) but we got errors when running our input only test cases. The classes that we looked into but did not carry on with are listed alphabetically as follows: ArrayList.java, Arrays.java, BigInteger.java, Dictionary.java, HashMap.java, Hashtable.java, LinkedList.java, TreeMap.java and URI.java.

Chapter 5

Results and Discussion

In this chapter we provide a complete report of our results. We first examine the effectiveness of the "null oracle" (Java runtime system) at detecting faults and then move on to the detection effectiveness of JML assertions. Next we classify JML assertions based on their effectiveness at different thresholds. We conclude the chapter with a discussion based on our results.

5.1 Effectiveness of the null oracle

Before evaluating the effectiveness of JML assertions at detecting errors, we first examine the efficacy of the null oracle at catching bugs. 759 valid mutants were generated for our seven subjects. Out of the 759 valid mutants, the null oracle killed 159 (20.95%) of them. Table 5.1 represents a breakdown of the valid mutants generated by muJava and the number of mutants killed by the null oracle for our seven test subjects.

Class Name	Valid Mutants	Caught by the null oracle
Date	319	55 (17.24%)
HashSet	93	17 (18.28%)
Observable	21	2 (9.52%)
Stack	40	26 (65%)
TreeSet	42	7 (16.67%)
URL	140	16 (11.43%)
Vector	104	36 (34.62%)

Table 5.1: Number of valid mutants killed by the null oracle

Table 5.1 does not include the 1236 mutants generated by muJava for the method **parse(String)** of Date. Because of the sheer size of the number of mutants generated for the method parse(String), we were unable to manually determine whether the mutants were valid or invalid. Of those 1236 mutants, 643 (52.02%) of the mutants were caught by the null oracle. If we add the mutants generated for parse(String), of the (759 + 1236) = 1995 mutants, (159 + 643) = 802 (40.2%) are caught by the null oracle. If we were to evaluate the number of invalid mutants from the 1236 generated for parse(String), then after throwing out those mutants which were not caught, the percentage of the mutants killed by the null oracle would be higher than 40.2%.

5.2 Effectiveness of JML Assertions

After determining the number of mutants caught by the null oracle, we evaluated how many of the mutants (for methods that had been annotated) that were not caught by the null oracle were caught with JML assertions. Out of the 275 valid mutants that were not killed by the null oracle whose methods contained JML assertions, 142 (51.64%) of them were killed when we used JML assertions as the test oracle. We did not take into account mutant implementations whose methods were not annotated as we wanted to find how effective JML assertions are at finding bugs when they are

implemented. Table 5.2 represents a breakdown of the valid mutants which were not caught by the null oracle and of those, how many were caught by JML assertions for our seven test subjects.

Class Name	Valid Mutants not caught by the null oracle	Caught by JML Assertions
Date	52	51 (98.08%)
HashSet	42	24 (57.14%)
Observable	19	2 (10.53%)
Stack	14	7 (50%)
TreeSet	28	3 (10.71%)
URL	60	40 (66.67%)
Vector	60	15 (25%)

Table 5.2: Number of valid mutants killed by JML Assertions that were not killed by the null oracle

Figure 5.1 depicts the classification of our valid mutants. 759 valid mutants were generated, out of which 159 (20.95%) were caught by the null oracle. Of the remaining 600 that were not caught by the null oracle, 325 did not contain annotations. Hence we only looked at (600 - 325) = 275 valid mutants that were not caught by the null oracle which did contain annotations to evaluate the effectiveness of JML assertions. Of the 275, JML assertions was successful in killing 142 (51.64%) mutants. The remaining 133 (48.36%) mutants were not killed by JML assertions.

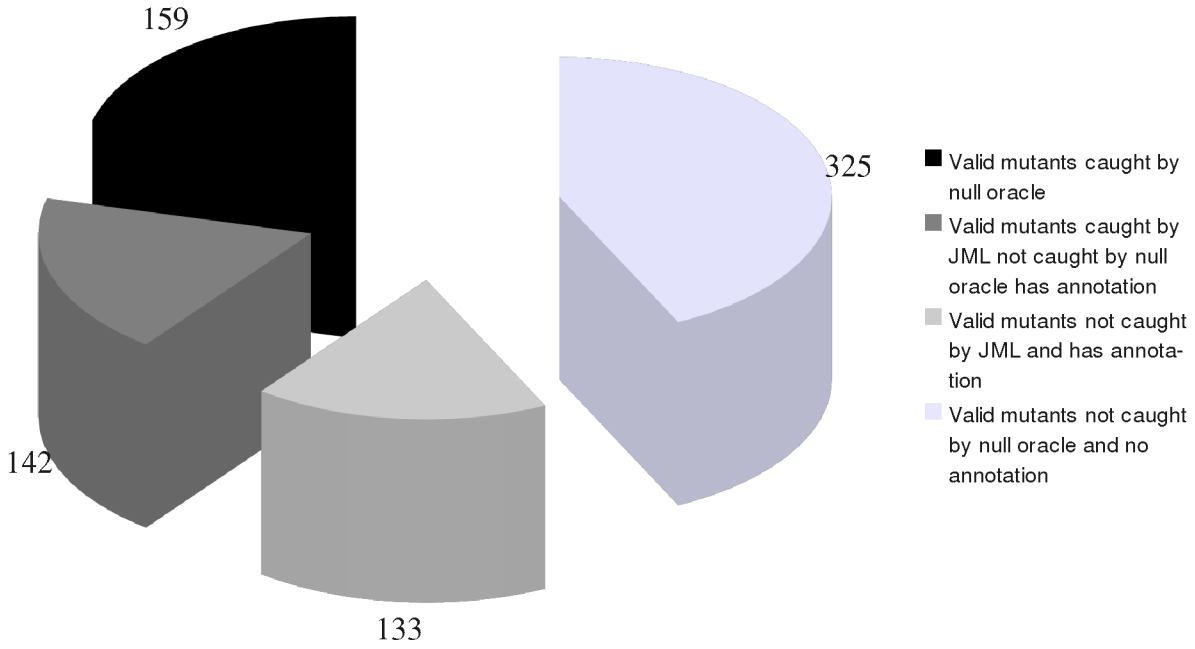

Figure 5.1: Valid Mutants distribution

5.3 Breakdown of our Results listed by Subject

For all our test subjects presented in Section 4.3, we provide a list of our results. As in Section 4.3, we provide a list of methods for each class along with the number of valid mutants for that method (Valid Mutants), the number of mutants caught by the null oracle (Caught by null oracle), the number of mutants not caught by the null oracle (Not caught by null oracle), the number of mutants caught as a result of using JML assertions as the test oracle (Caught by JML), the number of mutants not caught by JML assertions (Not caught by JML) and a JML caught weighted percentage (JML Caught Weighted %) that shows how effective JML assertions were at killing mutants that were not killed by the null oracle. We determine JML caught weighted % as follows:

$$\frac{(Caught\,by\,JML - Caught\,by\,the\,null\,oracle)}{(Valid\,mutants - Caught\,by\,the\,null\,oracle)} * 100$$

For methods that do not contain annotations, we put an N/A in the JML Caught Weighted % column.

5.3.1 Date

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
boolean after(Date)	6	0	6	6	0	100
boolean before(Date)	1	0	1	1	0	100
boolean equals(Date)	9	2	7	9	0	100
Date(int,int,int)	12	0	12	0	12	N/A
Date(int,int,int,int,int)	20	0	20	0	20	N/A
Date(int,int,int,int,int,int)	32	2	30	2	30	N/A
Date(long)	4	0	4	4	0	100
int compareTo(Date)	24	0	24	23	1	96
void setTime(long)	8	2	6	8	0	100
long getTimeImpl()	6	2	4	6	0	100

Table 5.3: Method-level results for Date

Table 5.3 lists the results we got for Date. Here, the null oracle only catches a few bugs where as JML assertions were really effective in killing the mutants that were not killed by the null oracle. All of the methods for Date that were annotated were simple methods with at most a couple of execution statements and branches.

5.3.2 HashSet

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
HashSet(int)	10	2	8	10	0	100
HashSet(int,float)	18	4	14	18	0	100
HashSet(Collection)	16	0	16	0	16	0
HashSet(int,float,boolean)	22	0	22	0	22	N/A
boolean add(Object)	2	0	2	2	0	100
boolean remove(Object)	2	0	2	0	2	0
void readObject(Object)	23	11	12	11	12	N/A

Table 5.4: Method-level results for HashSet

As in Date, the null oracle for HashSet only catches a few bugs while JML assertions proved quite effective here as well as listed in Table 5.4. Besides a couple

of methods **HashSet(Collection)** and **remove(Object)**, JML assertions killed all the remaining mutants.

5.3.3 Observable

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
void notifyObservers(Object)	17	0	17	0	17	0
void addObserver(Observer)	3	2	1	3	0	100
boolean hasChanged()	1	0	1	1	0	100

Table 5.5: Method-level results for Observable

For Observable, the null oracle hardly catches any bugs as depicted in Table 5.5. Although JML assertions catch a couple of bugs, it doesn't catch any for the method **notifyObservers(Object)**. We believe this is because we had to turn off certain test cases when testing Observable as mentioned in Subsection 4.3.3.

5.3.4 Stack

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
boolean empty()	0	0	0	0	0	0
int search(Object)	14	2	12	9	5	58
Object peek()	18	17	1	17	1	0
Object pop()	8	7	1	7	1	0

Table 5.6: Method-level results for Stack

Table 5.6 lists the results we got for Stack. The null oracle manages to kill a decent number of mutants here. However, JML assertions doesn't do as well. The methods listed for Stack are more complex and consist of four or more executable statements and multiple branches.

5.3.5 TreeSet

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
boolean add(Object)	2	0	2	2	0	100
boolean addAll(Collect	31	7	24	8	23	4
boolean remove(Object)	2	0	2	0	2	0
void readObject(Object)	7	0	7	0	7	N/A

Table 5.7: Method-level results for TreeSet

The null oracle as well as JML assertions seem to perform poorly when it come to TreeSet as shown by Table 5.7. Us having to turn off certain test cases as mentioned in Subsection 4.3.5 might be the reason behind this.

5.3.6 URL

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
boolean isValidProtocol(String)	68	4	64	4	64	N/A
int getPort()	4	0	4	4	0	100
HashSet(String,String,int,String)	4	2	2	4	0	100
URL(String,String,int,String,..)	56	10	46	37	19	59
URL(String,String,String)	1	0	1	1	0	100
void set(String,String,int,..)	7	0	7	6	1	86

Table 5.8: Method-level results for URL

Table 5.8 lists our results for URL. The null oracle only kills a few mutants here. However, JML assertions manage to kill a lot of mutants. For simple methods, JML assertions catch all the bugs but for complex methods with 10 or more statements and multiple branches, it only manages to kill 86 and 59% of the mutants that were not killed by the null oracle.

5.3.7 Vector

Method Name	Valid Mutants	Caught by null oracle	Not caught by null oracle	Caught by JML	Not caught by JML	JML Caught Weighted %
boolean add(Object)	4	4	0	4	0	0
boolean addAll(int,Collection)	12	7	5	7	5	0
boolean addAll(Collection)	4	2	2	2	2	0
boolean removeElement(Object)	0	0	0	0	0	0
int indexOf(Object,int)	2	2	0	2	0	0
int lastIndexOf(Object)	4	4	0	4	0	0
int lastIndexOf(Object,int)	2	1	1	1	1	0
Object lastElement()	4	0	4	3	1	75
Object remove(int)	12	3	9	8	4	56
Vector(Collection)	8	4	4	4	4	0
void addElement(Object)	4	0	4	0	4	0
void ensureCapacity(int)	0	0	0	0	0	0
void ensureCapacityHelper(int)	8	0	8	0	8	N/A
void insertElementAt(Object,int)	13	3	10	4	9	10
void removeAllElements()	1	1	0	1	0	0
void removeElementAt(int)	13	3	10	9	4	60
void removeRange(int,int)	12	1	11	1	11	0
void setSize(int)	1	1	0	1	0	0
void trimToSize()	0	0	0	0	0	0

Table 5.9: Method-level results for Vector

For Vector the null oracle performs quite well as shown in Table 5.9. JML assertions on the other hand, doesn't do as well. Besides four methods where it manages to kill some mutants, JML assertions do not kill any of the mutants not killed by the null oracle.

5.4 Evaluation of the Effectiveness of JML Assertions at Different Thresholds

The work done so far gave us insights into how effective the null oracle and JML assertions are when it comes to finding faults. However, with JML assertions, we also wanted to know how effective they were at different thresholds. Since we had already collected a lot of data, we wanted to utilize a machine learning tool and see if the tool could draw some conclusions from the data. We used Weka's J48 algorithm to help

us evaluate the effectiveness of JML assertions and come up with decision trees that fit our empirical data at different levels of effectiveness. We present the evaluation of JML assertions at 70, 80, 90 and 100% effectiveness below.

5.4.1 70% Effective

Figure 5.2 depicts a graphical representation of the decision tree created by Weka's J48 algorithm to determine when at least 70% of the bugs that were not killed by the null oracle are killed by using JML assertions as the test oracle. The oval nodes represent **decision nodes** and the rectangular light gray nodes represent **leaf nodes**. The numbers in (parenthesis) at the end of each leaf node tells us the number of instances in this leaf. If one or more leaves are not pure (i.e., all of the same class), the number of misclassified instances are given after a slash (/).

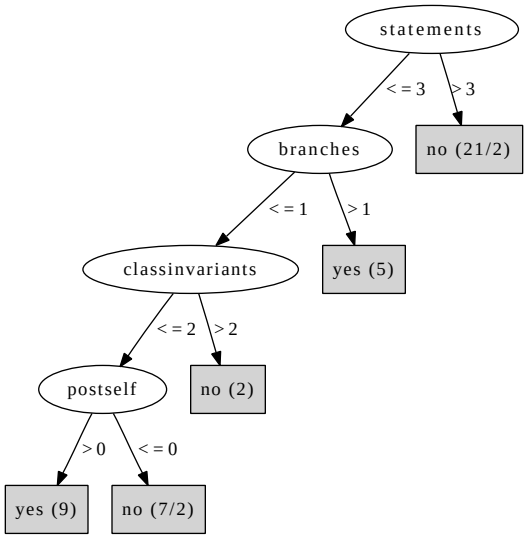

Figure 5.2: Evaluation of the effectiveness of JML Assertions at 70%

According to the decision tree in Figure 5.2, it is most likely that JML assertions will not be able kill more than 70% of the mutants for methods that have more than 3 statements (i.e., methods that are not simple). However, there are two misclassified

instances here. Another important decision node here is postself which is the number of postconditions defined for a method. The tree says that if a method has less than or equal to 3 statements, has less than or equal to one branch, has less than or equal to 2 class invariants and it has postconditions then bugs will be detected by JML assertions. When we do not have postconditions then the bugs will most likely not be caught. Here again, we have two misclassified instances.

```
> java weka.classifiers.trees.J48 -t data/Weka_Data/Grand_DataSet/Grand_70.arff

J48 pruned tree
------------------

statements <= 3
|   branches <= 1
|   |   classinvariants <= 2
|   |   |   postself <= 0: no (7.0/2.0)
|   |   |   postself > 0: yes (9.0)
|   |   classinvariants > 2: no (2.0)
|   branches > 1: yes (5.0)
statements > 3: no (21.0/2.0)

Number of Leaves  :     5

Size of the tree :     9

Time taken to build model: 0 seconds
Time taken to test model on training data: 0 seconds


=== Error on training data ===

Correctly Classified Instances          40               90.9091 %
Incorrectly Classified Instances         4                9.0909 %
Kappa statistic                          0.8053
Mean absolute error                      0.1472
Root mean squared error                  0.2713
Relative absolute error                 30.3984 %
Root relative squared error             55.174  %
Total Number of Instances               44


=== Confusion Matrix ===

  a  b   <-- classified as
 14  4 |  a = yes
  0 26 |  b = no


=== Stratified cross-validation ===

Correctly Classified Instances          31               70.4545 %
Incorrectly Classified Instances        13               29.5455 %
Kappa statistic                          0.3836
Mean absolute error                      0.3253
Root mean squared error                  0.4814
Relative absolute error                 66.9685 %
Root relative squared error             97.5994 %
Total Number of Instances               44


=== Confusion Matrix ===

  a  b   <-- classified as
 11  7 |  a = yes
  6 20 |  b = no
```

Figure 5.3: Output generated by Weka for 70% Effectiveness

Figure 5.3 shows the full output generated by Weka to determine when JML

assertions catch 70% of the bugs that the null oracle couldn't. The error on training data tells us that our classifier is 90.9% accurate (i.e, almost all the instances were classified correctly). The stratified cross-validation, which is a technique for assessing how the results of a statistical analysis will generalize to an independent data set, tells us that the accuracy of our classifier is 70.45%. From here onwards we will only provide the graphical decision tree and not the full output produced by Weka.

5.4.2 80% Effective

Figure 5.4 depicts a graphical representation of the decision tree created by Weka's J48 algorithm to determine when at least 80% of the bugs that are not killed by the null oracle are killed by JML assertions.

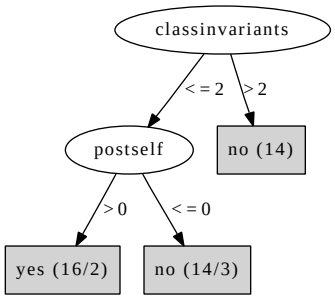

Figure 5.4: Evaluation of the effectiveness of JML Assertions at 80%

According to the decision tree in Figure 5.4, postconditions on self (abbreviated as postself in the graph) are important to catch 80% of the bugs. 30 out of the 44 total instances were classified based on the presence or absence of postconditions on self. The role of class invariants is a little misleading here as it seems to be counter intuitive. One would think that more class invariants would result in more bugs being caught, but all of our subjects did not have class invariants defined. Some classes that did not have any class invariants actually ended up killing more mutants whereas some classes that had 5 or more class invariants did not manage to kill as

many mutants. The error on training data for 80% effectiveness was 88.63% accurate and the stratified cross-validation was 79.54% accurate.

5.4.3 90% Effective

A graphical representation of the decision tree created by Weka to determine when at least 90% of the bugs that are not killed by the null oracle are killed by JML assertions is shown in Figure 5.5.

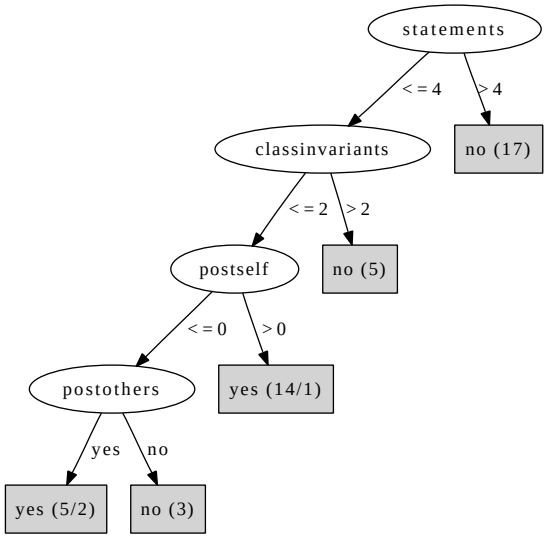

Figure 5.5: Evaluation of the effectiveness of JML Assertions at 90%

The decision tree in Figure 5.5 tells us that if a method is not simple (i.e., it consists of more than 4 statements) then it is really difficult to achieve 90% effectiveness with JML assertions (17 of the 44 instances classified). It also tells us that postconditions on self as well as postconditions on others (as described in Section 4.3) play an important role in killing mutants. For methods with less than 4 statements that had postcondition annotations, 14 out of 44 instances resulted in 90% of the mutants being killed. Here again we see some misleading information regarding class invariants the reasons for which are described in Subsection 5.4.2. The error on training data for

90% effectiveness was 93.18% accurate and the stratified cross-validation was 81.81% accurate.

5.4.4 100% Effective

Figure 5.6 depicts a graphical representation of the decision tree created by Weka to determine when all of the bugs that are not killed by the null oracle are killed by JML assertions.

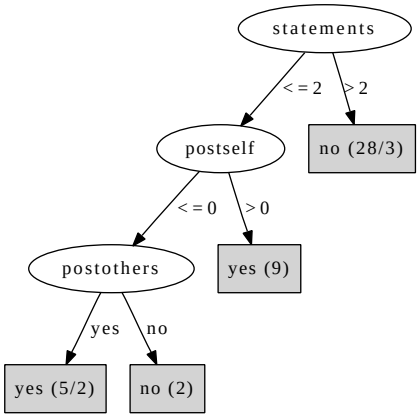

Figure 5.6: Evaluation of the effectiveness of JML Assertions at 100%

The decision tree in Figure 5.6 portrays that it is really hard to catch 100% of the bugs with JML assertions, unless the method we are dealing with is simple. For methods that consisted of more than 2 statements, in 28 of the 44 instances, all of the bugs were not caught. For methods with 2 or less than 2 statements, in order to catch 100% of the bugs, postconditions on self and postconditions on others were the most important factors with the presence of postcondition annotations on self resulting in 9 of the 44 instances being able to kill 100% of the mutants. The error on training data for 100% effectiveness was 88.63% accurate and the stratified cross-validation was 75% accurate.

5.5 Discussion

From our results we see that oracles play an important role in software testing. With the use of the null oracle, we were able to kill 159 (20.95%) of the 759 valid mutants. We describe in Section 5.1 how the percentage of the valid mutants killed by the null oracle should be around the region of 40.2%. However, this still doesn't catch more than half of the bugs.

With the use of JML assertions as the test oracle, we were able to kill 142 (51.64%) of the 275 valid mutants that were annotated. This tells us that assertions are effective at catching bugs but it does not however catch all the bugs. From our results we see that JML assertions are particularly useful in finding bugs for small non-complex methods. For complex methods JML assertions are not as useful. Also, writing assertions for complex methods can end up taking a lot of time as they require the use of model variables, model methods as well as ghost variables. The other observation that we were able to make from our results is that postcondition annotations, both on self and others, help in the detection of bugs. Hence, the presence of suitable postcondition annotations are a necessary means of detecting program faults.

With the help of Weka, we were also able to classify the effectiveness of JML assertions at different thresholds. We see that at 70% effectiveness, for methods that have more than 3 statements, JML assertions are not able to kill at least 70% of the mutants in 21 out of 44 instances. At 100%, for methods that have more than 2 statements, JML assertions are not able to kill all the mutants in 28 out of 44 instances. Another interesting statistic is for the postcondition on self annotation. There are 16 out of 44 instances where the presence of postcondition on self kills 80% of the mutants where as there are only 9 out of 44 instances where the presence of postcondition on self kills 100% of the mutants.

Chapter 6

Conclusion

In this chapter we provide our conclusions. Section 6.1 consists of an overview of the experience gained after finishing the experiment. In Section 6.2 we list a summary of our major findings and in Section 6.3 we describe our plans for future work.

6.1 Reaction

Oracles are an important part of the software testing process. Null oracles although useful, cannot help us catch most of the bugs. We need oracles to help us find the remaining bugs. The use of assertions as test oracles are also useful and can help in bug detection. However, assertions are only useful as test oracles in certain cases. They are more useful when dealing with simple methods but they cannot be relied upon to catch all the bugs when the unit under test is complex. When testing complex systems, we need to augment assertions with other approaches to test oracles.

6.2 Summary of Major Findings

During the process of running our experiment, we found that the null oracle was able to detect a fair share of the valid mutants. Out of the 759 valid mutants, the null oracle was able to kill 159 (20.95%) of them. We also found that the percentage of mutants caught by the null oracle could have been as high as 40.2%, as justified in Section 5.1. The use of assertions in the form of JML annotations as a test oracle on the other hand was able to detect 142(51.64%) of the 275 valid mutants that were annotated and were not caught by the null oracle. The use of Weka helped us classify the effectiveness of JML assertions at various thresholds. We found that JML assertions were useful in detecting bugs for small non-complex methods. When it came to complex methods, JML assertions did not fare as well. We were also able to determine that the postcondition annotation both on self and others helped catch most of the bugs.

6.3 Future Work

We believe that there is still additional information that we can get out of all the data that we have collected. For our purposes, we just created a data set based on all of the information that we had collected and derived a decision tree from Weka's J48 classification algorithm. It would be interesting to see the results from Weka after separating the instances that contained class invariants from the instances that did not contain class invariants. How differently would those two data sets be classified? Would they still look fairly similar? What would that say about the importance of class invariants when evaluating the effectiveness of JML assertions? We could do something similar with model variables and model methods and see what that tells

us about the importance of the use of model methods and model variables to detect faults.

As part of our future work we would also like to extend the number of test subjects that we currently have. Apart from just having Java libraries as our subjects, we would like to incorporate a few additional subjects that would help us collect more data and verify our results and conclusions to date.

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN: 978-0-521-88038-1.

[2] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE) 2005*, pages 402–411, St. Louis, MO, USA, May 2005.

[3] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26:55–69, 2000.

[4] K. Arnout and R. Simon. The .net contract wizard: Adding design by contract to languages other than eiffel. *Technology of Object-Oriented Languages, International Conference on*, 0:0014, 2001.

[5] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR01-02, 2001.

[6] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - java with assertions. In *Proceedings of the 2001 Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001.

[7] B. Baudry, Y. L. Traon, and J.-M. Jezequel. Robustness and diagnosability of oo systems designed by contracts. In *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, page 272, Washington, DC, USA, 2001. IEEE Computer Society.

[8] B. W. Boehm, R. K. McClean, and D. B. Urfrig. Some experience with automated aids to the design of large-scale reliable software. In *Proceedings of the*

international conference on Reliable software, pages 105–113, New York, NY, USA, 1975. ACM.

[9] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2004. available at http://www.swebok.org/.

[10] J. H. Bowser. Reference manual for ada mutant opeartors. Technical report, Georgia Institute of Technology, 1988.

[11] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw. Pract. Exper.*, 33(7):637–672, 2003.

[12] T. Budd and F. Sayward. Users guide to the pilot mu-tation system. Technical report, Yale University, 1977.

[13] T. A. Budd and R. J. Lipton. Proving lisp programs using test data. In *Proceedings of the Workshop on Software Testing and Test Documentation*, pages 374–403, December 1978.

[14] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[15] M. Carrillo, J. G. Molina, E. Pimentel, and I. Repiso. Design by contract in smalltalk. *Journal of Object-Oriented Programming*, 9(7), 1996.

[16] Y. Cheon. A runtime assertion checker for the java modeling language. Technical report, Iowa State University, 2003.

[17] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.

[18] P. Corporation. Using design by contract[TM] to automate Java[TM] software and component testing. Technical report, 2003.

[19] I. D. Curcio. Asap—a simple assertion pre-processor. *SIGPLAN Not.*, 33(12):44–51, 1998.

[20] M. E. Delamaro and J. C. Maldonado. Proteum - a tool for the assessment of test adequacy for c programs: User's guide. In *In Proceedings of the Conference on Performability in Computing Systems*, pages 79–95, 1996.

[21] R. A. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[22] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[23] C. K. Duby, S. Meyers, and S. P. Reiss. CCEL: A metalanguage for c++. In *IN USENIX C++ CONFERENCE*, 1992.

[24] A. Duncan and U. Holzle. Adding contracts to java with handshake. Technical report, University of California, 1998.

[25] R. W. Floyd. Assigning meaning to programs. volume 19, pages 19–32. American Mathematical Society, 1967.

[26] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Part II, vol. I. Technical report 1, 1947.

[27] P. Guerreiro. Simple support for design by contract in c++. *Technology of Object-Oriented Languages, International Conference on*, 0:0024, 2001.

[28] J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[29] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[30] J. M. Hanks. Testing cobol programs by mutation. Technical report, Georgia Institute of Technology, 1980.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[32] D. Hoffman. A taxonomy for test oracles. In *11th International Software Quality Week*, 1998.

[33] D. Hoffman. Heuristic test oracles. *Software Testing and Quality Magazine*, 1999.

[34] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification i: A logical basis and its implementation. Technical report, Stanford, CA, USA, 1973.

[35] S. K. John, J. A. Clark, and J. A. Mcdermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11:207–225, 2001.

[36] M. Karaorman, U. Hölzle, and J. Bruno. jcontractor: A reflective java library to support design by contract. In *In Proceedings of Meta-Level Architectures and Reflection, volume 1616 of lncs*, pages 175–196, 1999.

[37] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing, 1991.

[38] R. Kramer. icontract - the java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[39] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 204–212, Washington, DC, USA, 2006. IEEE Computer Society.

[40] G. T. Leavens, G. T. Leavens, A. L. Baker, A. L. Baker, C. Ruby, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. Technical report, Department of Computer Science, Iowa State University, 1999.

[41] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*. 2003.

[42] D. Luckham and F. Von Henke. An overview of anna, a specification language for ada. *Software, IEEE*, 2(2):9–22, March 1985.

[43] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering*, pages 352–363, Nov. 2002.

[44] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[45] M. A. Marin. Effective use of assertions in c++. *SIGPLAN Not.*, 31(11):28–32, 1996.

[46] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[47] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct 1992.

[48] M. Muller, R. Typke, and O. Hagner. Two controlled experiments concerning the usefulness of assertions as a means for programming. *Software Maintenance, 2002. Proceedings. International Conference on*, pages 84–92, 2002.

[49] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.

[50] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. pages 34–44, 2001.

[51] J. Offutt, P. Ammann, and L. L. Liu. Mutation testing implements grammar-based testing. In *MUTATION '06: Proceedings of the Second Workshop on Mutation Analysis*, page 12, Washington, DC, USA, 2006. IEEE Computer Society.

[52] D. Peters, S. Member, I. David, L. Parnas, and S. Member. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24:161–173, 1998.

[53] M. Pezz and M. Young. Testing object oriented software. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 739–740, Washington, DC, USA, 2004. IEEE Computer Society.

[54] R. Ploesch and J. Pichler. Contracts: From analysis to c++ implementation. *Technology of Object-Oriented Languages, International Conference on*, 0:248, 1999.

[55] S. Porat and L. P. Fertig. Class assertions in c++. *J. Object Oriented Programming*, 8(2):30–37, 1995.

[56] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[57] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing java modeling language contracts with aspectj. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[58] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[59] L. G. Stucki and G. L. Foshee. New assertion concepts for self-metric software validation. *SIGPLAN Not.*, 10(6):59–71, 1975.

[60] R. A. D. T. A. Budd, R. J. Lipton and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, 1979.

[61] R. P. Tan and S. H. Edwards. Experiences evaluating the effectiveness of jml-junit testing. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, 2004.

[62] T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu, and C. K. F. Luk. Testing object-oriented industrial software without precise oracles or results. *Communications of the ACM*, 2006.

[63] A. M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory, Cambridge University.

[64] J. Voas and K. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 152–157, Nov 1994.

[65] D. Welch and S. Strong. An exception-based assertion mechanism for c++. *J. Object Oriented Programming*, 11(4):50–60, 1998.

[66] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[67] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham. Weka: Practical machine learning tools and techniques with java implementations. In *Proc ICONIP/ ANZIIS/ANNES99 Future Directions for Intelligent Systems and Information Sciences*, pages 192–196. Morgan Kaufmann, 1999.

[68] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the international conference on Reliable software*, pages 450–455, New York, NY, USA, 1975. ACM.

[69] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.