On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis

Abstract

Mutation testing has traditionally been used as a defect injection technique to assess the effectiveness of a test suite as represented by a "mutation score." Recently, mutation test tools have become more efficient, and industrial usage of mutation analysis is experiencing growth. Mutation analysis entails adding or modifying test cases until the test suite is sufficient to detect as many mutants as possible and the mutation score is satisfactory. The augmented test suite resulting from mutation analysis may reveal latent faults and provides a stronger test suite to detect future errors which might be injected. Software engineers often look for guidance on how to augment their test suite using information provided by line and/or branch coverage tools. As the use of mutation analysis grows, software engineers will want to know how the emerging technique compares with and/or complements coverage analysis for guiding the augmentation of an automated test suite. Additionally, software engineers can benefit from an enhanced understanding of efficient mutation analysis techniques. To address these needs for additional information about mutation analysis, we conducted an empirical study of the use of mutation analysis on two open source projects. Our results indicate that a focused effort on increasing mutation score leads to a corresponding increase in line and branch coverage to the point that line coverage, branch coverage and mutation score reach a maximum but leave some types of code structures uncovered. Mutation analysis guides the creation of additional "common programmer error" tests beyond those written to increase line and branch coverage. We also found that 74% of our chosen set of mutation operators is useful, on average, for producing new tests. The remaining 26% of mutation operators did not produce new test cases because their mutants were immediately detected by the initial test suite, indirectly detected by test suites we added to detect other mutants, or were not able to be detected by any test.

1. Introduction

Mutation testing is a type of fault-based testing methodology in which the same test cases are executed against two or more program mutations (mutants for short) to evaluate the ability of the test cases to detect differences in the mutations (IEEE, 1990). Mutation testing has traditionally been used as a defect injection technique to assess the effectiveness of a test suite as represented by a "mutation score." Mutation testing can be thought of as "testing the test suite." Mutation *analysis* additionally entails adding or modifying test cases until the test suite is sufficient to detect as many mutants as possible (Murnane, et al., 2001). The augmented test suite resulting from mutation analysis may reveal latent faults and provides a stronger test suite to detect future errors which might be injected.

Mutation testing tools have recently become more efficient, and mutation analysis as a means for guiding the augmentation of an automated test suite is experiencing increased usage in the industrial community (Irvine, et al., 2007). Many software engineers look for guidance on how to augment their test suite via information provided by line and/or branch coverage tools (Krishnamurthy and Mathur, 1996). A test suite which covers more lines of the source is generally more likely to detect a fault (Zhu, et al., 1997). If the use of mutation analysis for test augmentation grows, then software engineers will want to know how the emerging technique compares with and/or complements coverage analysis. Additionally, software engineers can benefit from additional understanding about efficient mutation analysis techniques. To address this need for additional information about mutation analysis, we conducted an empirical study of the use of mutation analysis with a twofold purpose:

- To examine how mutation analysis compares with and/or complements coverage analysis as a means for guiding the augmentation of an automated test suite; and
- To help developers be more efficient in their use of mutation analysis by providing a summary of how many test-creating mutants an operator produces within a moderate sample of source code.

The study was conducted using two open source Java projects with corresponding JUnit[1] automated unit test cases: four versions for the Java backend of the iTrust[2] healthcare web application and HtmlParser[3]. The MuClipse[4] mutation testing plug-in for the Eclipse[5] integrated development environment was used. MuClipse was adapted from the MuJava (Offutt, et al., 2004) testing tool.

The remainder of this paper is organized as follows: Section 2 briefly explains mutation testing and summarizes other studies that have been conducted to evaluate its efficacy. Next, Section 3 provides information on MuClipse and its advancements for the mutation process. Section 4 details the test beds (i.e. the iTrust and HtmlParser projects) and the procedure used to gather our data, including terms specific to this study. Then, Section 5 shows the results, their interpretation, and the limitations of the study. Finally, Section 6 summarizes the results of our study in context.

2. Background and Related Work

First, Section 2.1 gives a definition of relevant terms and background information on mutation testing. Then, Section 2.2 reviews the concepts of fault injection. Finally, Section 2.3 presents related research.

2.1 Mutation Testing

A *mutant* (also program mutation, mutation) is a computer program that has been purposely altered from the intended version to evaluate the ability of test cases to detect the alteration (IEEE, 1990). Each mutant is a copy of

[1] http://junit.org/
[2] http://sourceforge.net/projects/iTrust
[3] http://sourceforge.net/projects/HtmlParser
[4] http://muclipse.sourceforge.net/

the original program with the exception of one atomic change. The test suite is then executed against this altered code to determine if the change can be detected. When any existing test case detects the mutant, the mutant is said to be "killed." Mutants that have yet to be detected are referred to as "living" (DeMillo, et al., 1988). A *mutation score,* calculated by taking the number of killed mutants and dividing by the total number of mutants, summarizes the results of mutation testing.

A *mutation operator* is a set of instructions for generating mutants of a particular type (Andrews, et al., 2005). For example, when a conditional mutation operator is used, a new copy of the implementation code is created and one instance of a binary operator is changed to another (e.g. changing `AND` to `OR`). Mutation operators are classified by the language constructs they are created to alter (e.g. method-level, class-level, exception). Traditionally, the scope of operators was limited to the method level (Alexander, et al., 2002). Operators which exclusively mutate statements within a method body are referred to as method-level operators and are also known as traditional operators. Recently class-level operators, or operators that test at the object level, have been developed (Offutt, et al., 2006). Certain class-level operators in the Java programming language, for instance, replace method calls within implementation code with a similar call to a different method. Class-level operators take advantage of the object-oriented features of a given language. They are employed to expand the range of possible mutation to include specifications for a given class and inter-class execution.

Most often, mutation operators produce mutants which demonstrate the need for more test cases (Frankl, et al., 1997). However, for various reasons that will be discussed, mutation operators may sometimes produce mutants which cannot be detected by adding more tests to the test suite. The developer must manually determine these mutants are *stubborn mutants*[6]. Stubborn mutants are those which cannot be killed due to logical equivalence or due to language constructs (Hierons, et al., 1999). We can relate stubborn mutants in mutation testing to false positives in static analysis (Hovemeyer and Pugh, 2004) because both cause excess system processing and the software engineer to expend energy when no problems exist (e.g. no additional test case is needed and no fault exists, respectively). Alternatively, the process of adding a new test case sometimes kills more than one mutant. Two mutants dying by the execution of one test case indicates that one of the mutants might not have provided any useful information beyond that provided by the other. Testing efficiency can be gained by not using mutation operators which consistently reveal the same test deficiency. Stubborn and similar mutants can cause the mutation process to be computationally expensive and inefficient (Frankl, et al., 1997). As a result, empirical data about the behavior of the mutants produced by a given mutation operator can help us understand the utility of the operator in a given context.

[5] http://www.eclipse.org

[6] *Stubborn* mutants are more clearly defined as those living mutants that may or may not be *equivalent* to the original source code. Sometimes, a mutant remains alive and yet cannot be feasibly proven equivalent through formal analysis (Hierons, et al., 1999).

2.2 Fault Injection

With fault injection, a software unit is assessed for its validity. Subsequently, a modified copy of the implementation code is created to contain one or more changes based on common programmer errors and/or changes which may have a major impact on the way components in the system interact (Voas and Miller, 1995). The software is then re-assessed to determine whether the injected fault can be detected via a validation and/or verification technique. The procedure of injecting these faults is based on two ideas: the Competent Programmer Hypothesis and the Coupling Effect. The Competent Programmer Hypothesis states that developers are generally likely to create a program that is close to being correct (DeMillo, et al., 1978). The Coupling Effect assumes that a test built to catch a small change (generally in one code location) will be adequate to catch the ramifications of this atomic change on the rest of the system (DeMillo, et al., 1978). The modified copy of the implementation code (or program mutant) can then be used for two purposes: 1) to determine regions of code which propagate errors; or 2) to determine the ability of the validation and/or verification method to detect errors which may be in the system or which are currently in the system (Voas and Miller, 1995). Mutation testing can be thought of as automated fault injection and serves the purpose of the latter.

2.3. Related Studies

Offut, Ma and Kwon contend, "Research in mutation testing can be classified into four types of activities: (1) defining mutation operators, (2) developing mutation systems, (3) inventing ways to reduce the cost of mutation analysis, and (4) experimentation with mutation" (Offutt, et al., 2006). In this sub-section, we summarize the research related to the last item, experimentation with mutation, the body of knowledge to which our research adds.

Several researchers have investigated the efficacy of mutation testing. Andrews, et al. (Andrews, et al., 2005) chose eight well-known C programs to compare hand-seeded and naturally-occuring faults to those generated by automated mutation engines. The authors found that mutation testing produced faults (in the form of mutants) which were statistically representative of real-world faults, and therefore the mutation testing process is a reliable way of assessing the fault-finding effectiveness of a test suite. The authors additionally compared their hand-seeded faults and automatically-generated mutants to naturally-occurring faults in one program. Statistically, the hand-seeded faults and mutants were no harder to detect than the naturally-occurring faults, which supports the Competent Programmer Hypothesis. The authors warn that the results of their study may not necessarily generalize. Specifically, the authors indicate that manually-seeded faults may not be similar to those found in industrial applications, and although one of their subject programs contained naturally occurring faults, the others had to be manually produced by "experienced engineers" (Andrews, et al., 2005). The authors contend that the experiment should be replicated with more subject programs containing naturally-occurring faults to add experimental validity to their claims.

Andrews, et al. (Andrews, et al., 2006) furthered the previously-mentioned study by performing an empirical study on one industrial program with known system testing faults. The authors used mutation analysis to assess whether other testing criteria such as control and data flow are sufficient to determine the fault-finding ability of a test suite. The authors found that mutation testing seeds faults which resemble those found naturally in industrial environments, again supporting the Competent Programmer Hypothesis. The authors contend that, based on their results, mutation testing can be used to accurately determine the fault-finding effectiveness of a testing methodology. In comparing block, decision, c-use and p-use coverage criteria, the authors discovered that mutation testing rates each of these as equally cost-effective with high statistical significance. However, the authors point out that creating a test suite to satisfy each of these increasingly more strict criteria does cause that test suite to be more effective at finding faults.

Frankl and Weiss (Frankl, et al., 1997) compare mutation testing to all-uses testing by performing both on ten small (always less than 78 lines of code) C programs. The authors inspected and recorded the locations of naturally-occurring faults within these programs. All-uses testing generates a test suite which will traverse every possible path through the call-graph. The authors concede that for some programs in their sample population, no all-uses test suite exists. The results were mixed. Mutation testing proved to uncover more of the known faults than did all-uses testing in five of the nine case studies, but not with a strong statistical correlation. The authors also found that in several cases, their tests killed every mutant but did not detect the naturally-occurring fault, indicating that high mutation score does not always indicate a high detection of faults.

Offut et al. (Offutt, et al., 1996) also compare mutation and all-uses testing. Their chosen test bed was a set of ten small (less than 29 lines of code) Fortran programs. The authors chose to perform cross-comparisons of mutation and data-flow scores for their test suites. After completing mutation testing on their test suites by killing all non-stubborn mutants, the test suites achieved a 99% all-uses testing score. After completing all-uses testing on the same test suites, the test suites achieved an 89% mutation score. The authors do not conjecture at what could be missing in the resultant all-uses tests.

Additionally, to verify the efficacy of each testing technique, Offut, et al. (Offutt, et al., 1996) inserted 60 faults into their source which they view as representing those faults that programmers typically make. Mutation testing revealed on average 92% of the inserted faults in the ten test programs (revealing 100% of the faults in five cases) whereas all-uses testing revealed only 76% of inserted faults on average (revealing 100% of the faults in only two cases). The range of faults detected for all-uses testing is also significantly wider (with a range of 15-100%) than that of mutation testing (with a range of 67-100%).

Ma et al. (Ma, et al., 2006) conducted two case studies to determine whether class-level mutants result in a test suite which detects more faults. The authors used MuJava to mutate BCEL, a popular byte code engineering library, and collected data on the number of mutants produced for both class-level and method-level. The authors mutated the library using a technique known as selective mutation, which uses a subset of the method-level operators to reduce execution cost while maintaining effectiveness. Mutant generation using class-level operators revealed that most Java classes were mutated by at least one class-level mutation operator. This result indicates that object-oriented features, such as the `static` keyword, can be found throughout the system under test. However, most operators applied to a small percentage of the classes (10% or so). This result indicates that the *same* object-oriented feature is not found in a high proportion of classes in the system under test. Whereas we may see the use of the `static` keyword in one class, in another class the object-oriented feature involved may be the `this` keyword.

Additionally, Ma et al. (Ma, et al., 2006) mutated their subjects and killed every non-stubborn method-level mutant and ran the resultant test set against the class-level operators. The outcome demonstrated that at least five of the class-level mutation operators produce mutants with high mutation scores (>50%). These high mutation scores indicate that these operators may not produce many new tests since their mutants were killed by test sets already written to kill method-level mutants. The study also revealed that two class-level operators (EOA and EOC) resulted in a 0% mutation score, indicating that these operators could be a positive addition to the method-level operators. However, the authors concede that the study was conducted on one sample program, and thus these results may not be representative.

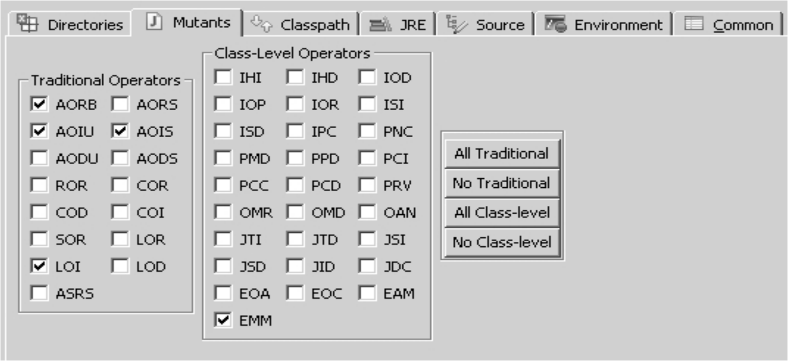
Namin and Andrews (Namin and Andrews, 2006) describe an experimental process for deciding on a set of mutation operators for C. Using seven C programs with an average of 326 non-whitespace/commented lines of code, the authors executed the mutant generator Proteum which contains 108 mutation operators. Namin and Andrews then use a set of variable reduction techniques (such as correlation analysis, principal component analysis and cluster analysis) from statistics to determine the adequate mutation operator set. A correlation value was calculated which was expressed for two mutation operators A and B as the number of test cases which kill a set of mutants from operator A and a set of mutants from operator B. Variable reduction then uses algorithms which sort and prioritize mutation operators based on these correlation values. At the time of publication, Namin and Andrews had yet to conduct their experimental analysis, but contend that they have proposed a statistical methodology for determining the sufficient mutation operator set.

We have conducted an earlier study (Smith and Williams, 2007) using MuClipse with a smaller dataset and different analysis methodology. The purpose of this earlier study was to classify the behavior of mutation operators as a whole and to look for patterns in the implementation code which remain uncovered by the test set after attempting to kill all mutants. In this earlier study, we used the same classification scheme for our mutants as can be found in Section 4.4. We found that for the back-end of the iTrust healthcare web application, mutation operators which alter conditional expressions were the most effective at producing new test cases, and operators which alter arithmetic operations were the least effective, as they produced mostly stubborn mutants. We also found that after attempting to kill all mutants, there were still a number of lines of production code which were unexecuted by the resultant test set and that most of these lines of code fell within a Java catch block.

3. MuClipse

Offut, Ma and Kwon have produced a Java-based mutation tool, MuJava (Offutt, et al., 2004), which conducts both automated subtasks of the mutation process in Java 1.4: generating mutants and running tests against created mutants. For generating mutants, MuJava provides 16 method-level and 29 class-level mutation operators. The method-level operators were chosen based on the concept of a “selective set,” which would effectively execute mutation testing with the fewest number of operators (Offutt, et al., 1996). Developers can choose which operators will be used to generate mutants and where mutant source files will be placed. MuJava requires unit tests in a Java 1.4 class containing public methods which include the word "test" in their signature. MuJava stores and displays the return values from these test methods (any Java primitive) in the console window during original result collection and mutant result collection. The mutation score and number of detected mutants are displayed at the end of the test execution process.

The MuJava application provided the base for the development of MuClipse[4], an Eclipse plug-in created by the first author. MuClipse provides the developer with an application, integrated with the development environment, which both generates and tests mutants in Java 1.4. In generating mutants, MuClipse allows the developer to select the mutant operators to employ and which classes should be mutated (see Figure 1).



Traditional Operators

☑ AORB ☐ AORS
☑ AOIU ☑ AOIS
☐ AODU ☐ AODS
☐ ROR ☐ COR
☐ COD ☐ COI
☐ SOR ☐ LOR
☑ LOI ☐ LOD
☐ ASRS

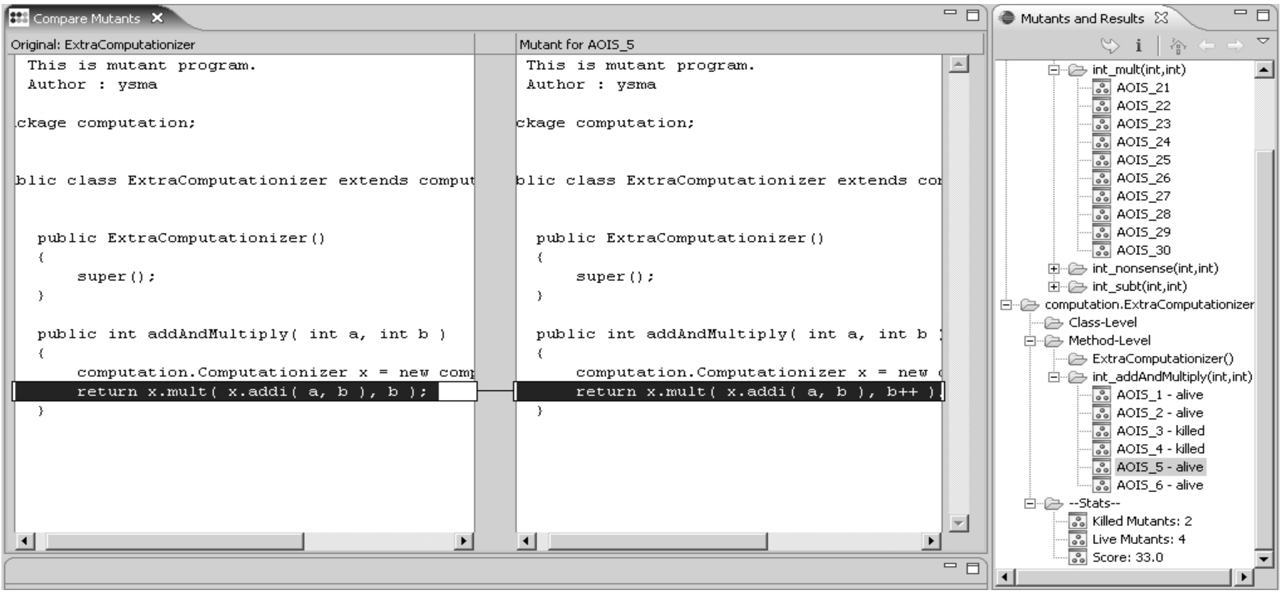
Class-Level Operators

☐ IHI ☐ IHD ☐ IOD
☐ IOP ☐ IOR ☐ ISI
☐ ISD ☐ IPC ☐ PNC
☐ PMD ☐ PPD ☐ PCI
☐ PCC ☐ PCD ☐ PRV
☐ OMR ☐ OMD ☐ OAN
☐ JTI ☐ JTD ☐ JSI
☐ JSD ☐ JID ☐ JDC
☐ EOA ☐ EOC ☐ EAM
☑ EMM

Figure 1. Selecting Operators in MuClipse

When testing mutants, MuClipse allows the developer to decide which class's mutants are to be tested; which test suite is to be run against chosen mutants; and which type of mutant (class- or method-level) operators should be run. The mutation operators appearing in Figure 1 under the panel labeled "Traditional Operators" are what we have thus far referred to as method-level operators. MuClipse allows developers the choice of using JUnit[1] test cases as well as MuJava test cases when attempting to kill resultant mutants. JUnit test cases inherit functionality from an abstract TestCase object to provide a result of pass, fail or error based on the oracles the tester chooses. Developers use JUnit `assert` statements, which are polymorphic to provide the tester the ability to assert that a given runtime variable or return value contains a desired static result, as in `assertEquals(3, Parser.getLinks().getSize());`. When the assertion does not hold, a failure is recorded. Otherwise, the test continues to execute. MuClipse stores these results as a Boolean true, false or Java Exception when collecting original results or mutant results.

Another major component of the mutation process is the management of mutants and their statuses. MuClipse implements an integrated Eclipse View (see Figure 2) which displays each mutant and its status, organized by Java class and producing operator. This View also contains the overall statistics for living and killed mutants, and the calculated mutation score.



```
 This is mutant program.
 Author : ysma

ckage computation;


blic class ExtraComputationizer extends comput


  public ExtraComputationizer()
  {
      super();
  }

  public int addAndMultiply( int a, int b )
  {
      computation.Computationizer x = new comp
      return x.mult( x.addi( a, b ), b );
  }
```

```
 This is mutant program.
 Author : ysma

ckage computation;


blic class ExtraComputationizer extends cor


  public ExtraComputationizer()
  {
      super();
  }

  public int addAndMultiply( int a, int b )
  {
      computation.Computationizer x = new c
      return x.mult( x.addi( a, b ), b++ )
  }
```

```
      int_mult(int,int)
         AOIS_21
         AOIS_22
         AOIS_23
         AOIS_24
         AOIS_25
         AOIS_26
         AOIS_27
         AOIS_28
         AOIS_29
         AOIS_30
      int_nonsense(int,int)
      int_subt(int,int)
computation.ExtraComputationizer
   Class-Level
   Method-Level
      ExtraComputationizer()
      int_addAndMultiply(int,int)
         AOIS_1 - alive
         AOIS_2 - alive
         AOIS_3 - killed
         AOIS_4 - killed
         AOIS_5 - alive
         AOIS_6 - alive
   --Stats--
      Killed Mutants: 2
      Live Mutants: 4
      Score: 33.0
```

Figure 2. Comparing Mutants to Originals in MuClipse

Some design decisions were made in the MuJava-to-MuClipse modification to increase the efficiency of MuClipse test execution. MuClipse stores living and killed mutant names in a file, such that when mutants are killed, they are not executed in the current or future sessions. Secondly, MuClipse only gathers the results for the test case on the original code once and stores them for comparison during the rest of the current session of the mutation process. Thirdly, some classes contain methods which MuJava is designed not to mutate, such as the Java `main` method. MuClipse does not gather test results for a method within a class that contains no mutants.

4. Research Method

Previous research has shown that mutation testing is an effective way to assess the ability of test cases to detect injected faults. As stated in the introduction, we have performed mutation analysis on two open source projects' production code for the following objectives:

- To examine how mutation analysis compares with and/or complements coverage analysis as a means for guiding the augmentation of an automated test suite; and
- To help developers be more efficient in their use of mutation analysis by providing a summary of how many test-creating mutants an operator produces within a moderate sample of source code.

These high-level objectives are further refined into more precise, lower-level research questions:

1) What effect does performing mutation analysis have on the line and branch level coverage of an augmented test suite?
2) Is there a relationship between the mutation operators chosen for mutation analysis and the number of new test cases?

3) Which mutation operators consistently produce the most additional test cases?
4) Does every mutant result in a new test case?
5) Is there any way to evaluate mutation analysis or mutation operators for the objective of augmenting a test suite?

In Section 4.1, we explain terms used in this study which classify mutant results throughout the testing procedure. Next, Section 4.2 contains information on the design, requirements and architecture of one open source project in our test bed, the iTrust application. Section 4.3 details the same for HtmlParser, a second open source project in our test bed. Finally, Section 4.4 gives a step-by-step description of the procedure used to conduct our research.

4.1 Additional Classification Terms

Mutants have typically been classified as **killed**, **living** or **stubborn** (Hierons, et al., 1999), as used earlier in this paper. However, we consider it important not only that a mutant dies, but *when* it dies. Consider a mutant X (created by mutation operator `A`) that the developer attempts to kill using test case T. Consider that mutant Y (created by mutation operator `B`) is also killed upon the execution of test case T. Possibly the "two-for-the price-of-one" payoff of test case T may be an anomaly. Or alternatively, perhaps mutation operators `A` and `B` generate redundant mutants, or mutants that are often killed by the same test case(s). Previously living mutants killed by a test case written to kill other mutants are called **crossfire**. Since mutants represent possible faults, crossfire mutants also indicate that a developer trying to kill a single mutant will sometimes detect more injected faults than intended.

Crossfire mutants, at first glance, appear to be useless at producing effective test cases. However, a crossfire mutant could produce effective tests if it is encountered earlier in the mutation process. Our intuition is that there exists some number of mutants which would be classified as crossfire regardless of the order in which they were encountered. Crossfire mutants can be thought of as insurance that a given set of programming errors is tested. However, we could possibly reduce the number of mutants that result in the same test case. Usually, a single operator produces several mutants from a single match of a given regular expression within the implementation code.

Additionally, a mutant which dies on the first execution of test cases indicates the current set of tests were adequate and does not yield a new test case. Mutants that die at the first execution are called "Dead on Arrival" or **DOA.** Empirically, DOA mutants may indicate that the operator which created the mutants generates mutations which are detected via testing strategies typically used in software development practices, such as boundary value analysis and/or equivalence class partitioning.

The MuJava mutation engine (underlying MuClipse) does not operate on compiled binary Java classes. Instead, it alters the Java source code and compiles the result. Operating on implementation code can lead to two syntactically different expressions within Java being compiled to the same binary object. For example, if a local instance of any subclass of `java.lang.Object` is created, but not initialized within a class, the Java Virtual

Machine automatically initializes this reference to null. Though developers find automatically-initialized variables convenient, the construct causes the code snippets in Figure 3 to be logically equivalent. No test case can discern the difference between a variable which was initialized to null due to the Java compiler and a variable which was explicitly initialized to null by the developer, causing the mutant to be **stubborn**.

The definition of stubborn is used in the previously described sense. We added definitions for DOA and crossfire, and we alter the definition of killed for the context of our study. In summary, we use the following terms to classify generated mutants:

- **Dead on Arrival (DOA).** Mutant that was killed by the initial test suite.
- **Killed.** Mutant which was killed by a test case which was specifically written to kill it (that was not in the initial test suite).
- **Crossfire.** Mutant that was killed by a test case intended to kill a different mutant.
- **Stubborn.** Mutant that cannot be killed by adding an additional test case.

Killed mutants provide the most useful information for our empirical study: additional, necessary test cases. DOA mutants might be considered the easiest to kill since they were killed by the initial test. However, the initial test set varies substantially in fault-detection ability from program to program. Therefore, DOA mutants provide an indication of the fault-detection ability of the test suite at the outset of the mutation process. Crossfire mutants indicate that our tests are efficient at detecting sets of related errors and may indicate redundant mutants. An operator that has a history of producing a high percentage of stubborn mutants may be a candidate for not being chosen for mutant generation.

4.2 iTrust Application

iTrust is an open source web-based healthcare application that has been created and evolved by North Carolina State University students. We chose to perform our experiment on iTrust because few studies thus far have focused on how mutation testing performs when used with a web-based application. The motivation for the creation of iTrust was to provide a course project for use in learning the various types of testing and security measures currently available. Several teams created versions of the application in a Software Testing and Reliability course in the Fall

```
//the original code
ITrustUser loggedInUser = null;

//the mutated code
ITrustUser loggedInUser;
```

Figure 3. Logically Equivalent Code

of 2005. The best team's code (released as iTrust v1) was then enhanced by seven two-person teams in the same course in the Fall of 2006. All teams were given the same requirements and the same starting source code, but were not held to any design specification. Throughout the semester, the teams were instructed to unit test their source code and attempt to maintain an 80% line coverage score using techniques such as equivalence class partitioning and boundary value analysis. The teams did not always meet this requirement. For example, we found many test cases within our starting set executed procedures in iTrust without any corresponding `assert()`, or oracle statements, to artificially improve coverage scores. JUnit does not require an `assert()` statement with every test case. However, if a test case calls a method in the iTrust code, these lines are recorded as covered whether or not their return value is verified. We removed all test cases without an assert statement from the preexisting test suite.

We randomly chose four of these seven teams (which we will call Teams A-D) and conducted mutation analysis on three classes of their iTrust projects (known as iTrust v2a-d). For our mutation analysis study, we chose the classes of iTrust (`Auth`, `Demographics` and `Transactions`) that performed the bulk of the computation and dealt directly with the database management back-end using SQL. The other classes in the framework act primarily as data containers and do not perform any computation or logic. Complexity metrics[7] for each Java class in the iTrust framework for Teams A-D are in Table 1 and Table 2.

Table 1. Line, Field, Method Counts for iTrust v2a-d for Java package edu.ncsu.itrust.

	Line Count for Team				**Field Count for Team**				**Method Count for Team**			
Class	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
Auth	280	299	278	278	2	2	2	2	16	16	16	16
Demographics	628	544	828	540	27	25	25	25	22	18	26	19
Transactions	123	120	133	183	2	2	2	2	7	7	7	10

Table 2. LoC and Class Counts for iTrust v2a-d in Java package edu.ncsu.itrust.

Team	**A**	**B**	**C**	**D**
LoC Tested	1031	963	1239	1001
Total (for all classes, tested and not tested)	2295	2636	3448	2867
Number of Implementation Classes	20	18	21	26
Number of JUnit Classes	8	5	13	41

iTrust was written in Java 1.5 using a Java Server Pages (JSP)[8] front-end. Because the model for the application was to limit the use of JSP to the graphical user interface code, the three primary classes of the iTrust framework support the logic and processing, including database interaction for the iTrust application. `Auth` is used in each group's project to authenticate and manage users and enforce role-based access to iTrust, and as a result can instantiate `Demographics` and `Transactions` with a specified logged in user. Otherwise, there are no dependencies between the three classes.

[7] LoC is every line within the Java implementation code file which is not 1) blank or 2) a comment. LoC calculated using NLOC: http://nloc.sourceforge.net/. Methods refers to public, private, protected, static methods and constructors. Fields refers to public, private, protected and static local variables (declared within the class definition).

[8] http://java.sun.com/products/jsp/

4.3 HtmlParser Application

HtmlParser is a Java 1.4 open source library used to represent HTML and XML tags as Java objects in a linear or tree-based fashion. We chose to use HtmlParser in our study because it was created by many unassociated developers in an open source environment and because it is compatible with the mutation engine underlying MuClipse. The development team for HtmlParser has maintained corresponding JUnit tests for their project, though we are not sure by what methodology or to what coverage criteria the team was held. Complexity metrics for HtmlParser v1.6 can be found in Table 3 and Table 4. Its authors state that HtmlParser is typically used for two use cases: Extraction and Transformation. Extraction covers uses such as link validation, screen scraping and site monitoring. Transformation covers uses such as website capturing, syntactical error checking, and HTML to XHTML conversion. We chose three subject classes from HtmlParser which

- were of similar size (average 315 LoC) to those chosen from the iTrust framework;
- contained Javadoc indicating they were likely to be covered in a typical use case (based on the project description); and
- had directly corresponding JUnit test cases.

Table 3. Line, Field and Method Counts for HtmlParser v1.6

Package	Class	Line Count	Field Count	Method Count
org.htmlparser	Attribute	263	4	28
	Parser	318	8	34
org.htmlparser.beans	StringBean	365	19	24

Table 4. LoC and Class Counts for HtmlParser v1.6

Total Tested	946
Total (for all classes, tested and not tested)	21801
Number of Implementation Classes	148
Number of JUnit Classes	80

Attribute is a container which stores the name, assignment string, value and a single quotation mark using getters and setters for the attribute of an HTML tag in a Java object (e.g. as in "href='index.html'"). Information about tag attributes can then be manipulated or transliterated in the form of a Java object.

Parser is the main class for the HtmlParser library. Parser, when given a URL or an input string, can return a list of nodes representing all HTML tags within this resource as Java objects. Alternatively, Parser can return a list of these objects based on a provided filter or visitor.

StringBean uses the Visitor (Gamma, et al., 1995) pattern to extract strings (in this case, any textual information not representing an HTML tag) from an HTML source. StringBean also uses the Observer (Gamma, et al., 1995) pattern to update any registered class when its properties change. Properties supported include whether or not to strip white space from the source, whether to replace non-breaking space pages and whether to include link text in the output. StringBean contains an instance of Parser which acts as its HTML source and this instance is used throughout the class to extract the next HTML string. Otherwise, there are no dependencies between the three classes.

4.4 Experimental Mutation Testing Process

This section gives a step-by-step description of the procedure used to conduct our experiment.

4.4.1 Description

As shown in Figure 4, the first part of our experimental mutation testing process is to employ mutation operators (e.g. Operator1) to alter the code under test into several instances, called mutants (e.g. O1M1), using an automated mutation engine. The second part of the process (e.g. Test Execution) is to record the results of the test suite (e.g. TestCase1 and TestCase2) when it is executed against each mutant (e.g. O1M1: Killed). We use these test results to complete the third part of the process, inspection. Starting with a test suite with all passing results, the mutant is said to be *killed* (Alexander, et al., 2002) when the test suite contains any test which fails due to the mutation (O1M2: Killed). If every test in the suite passes with the mutation, then the mutant is said to be *living* (Alexander, et al., 2002) because the mutant introduced an error into the program

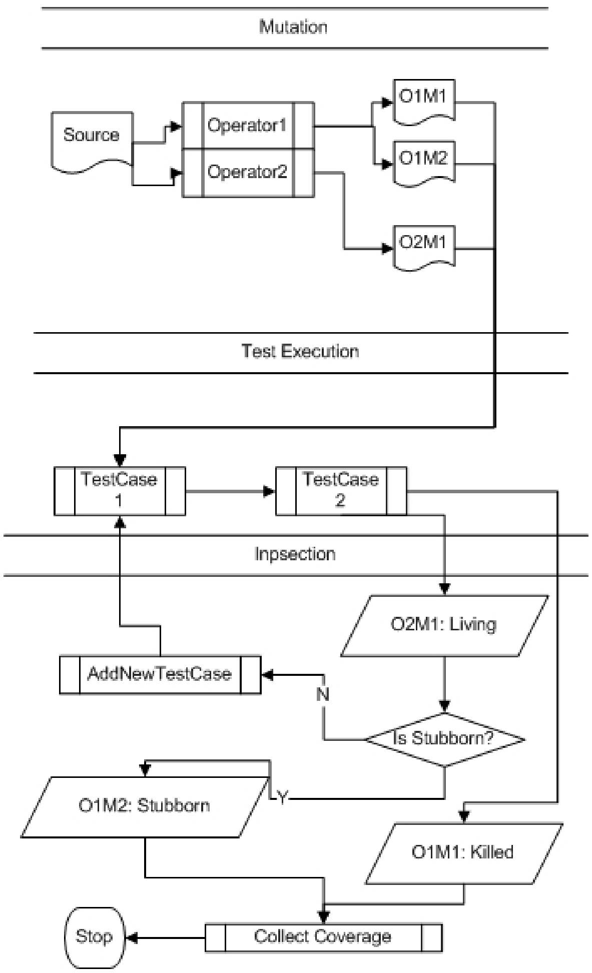


Figure 4. The Mutation Testing Process

that was not detected by the test suite (e.g. O2M1: Living). Augmenting the test suite with another test (e.g. NewTest) may kill living mutants. However, *stubborn* mutants (e.g. O1M2: Stubborn) produce no detectable difference in program execution (see Section 4.1). When attempting to kill all mutants, the inspection of a stubborn mutant does not produce a new, additional test case because a stubborn mutant cannot be detected (see Section 4.1); however, inspecting a living but killable mutant indeed produces a new test case. Since our study focuses on the ability of mutation analysis to augment the existing test set, we measure the number of mutants which fall into each of the categories defined in Section 4.1.

After each mutant has been inspected, a mutation score is calculated by dividing the number of killed mutants by the total number of mutants. A mutation score of 100% is considered to indicate that the test suite is adequate (Offutt, et al., 1996). However, the inevitability of stubborn mutants may make a mutation score of 100% unachievable. In practice, the mutation testing process entails creating a test suite which will kill all mutants that can be killed (i.e., are not stubborn).

We conducted an empirical evaluation of mutation analysis on two open source projects: four versions of the three core classes of the iTrust application (see Section 4.2) and three selected classes from HtmlParser (see Section 4.3) that had associated JUnit test suites. For each class, class-level and method-level mutants were generated using all available operators provided by the MuJava framework (which underlies MuClipse). Testing and generation of mutants were executed using Eclipse v3.1 on an IBM laptop with a 1.69 Ghz processor and 1.5 GB of RAM running Microsoft Windows XP. Eclipse and MuClipse were compiled and executed using Java 1.5. iTrust was written to comply with a SQL back-end and thus was configured to interface with a locally executing instance of MySQL 5.0[9]. HtmlParser does not interact with a database and thus we were not required to configure it in any way.

We used the following procedure to gather data about our mutation operators:

1. Execute the test cases against all generated mutants (a process provided by MuClipse as shown in Figure 4 under Mutation and Test Execution). Record the classification (e.g. living or killed) of each mutant and record the mutation score, line coverage and branch coverage. Record all mutants killed by the initial test execution as **Dead on Arrival (DOA, see Section 4.1)**.
2. Inspect the next (or first) living mutant by hand as it appears in the "View Mutants and Results" control in MuClipse (see Figure 2, O2M1: Living). If there are no living mutants or no mutants at all, proceed to Stop.
3. If this mutant cannot be killed (see Section 2.1), record it as **stubborn** and return to Step 2.
4. Write a test case intended to kill this and only this mutant (see Figure 4, NewTestCase).
5. Execute the test suite, including the newly-created test case, against the remaining living mutants (see Figure 4, Test Execution). Since the newly added test case actually does kill the mutant in question, record the mutant as **killed**. Record other mutants that are killed by this newly created test case as **crossfire (see Section 4.1)**.
6. Execute the test suite and record the line and branch coverage.

[9] http://www.mysql.com/

7. If there are no more living, non-stubborn mutants then Stop (see Figure 4, Stop). Otherwise, proceed to Step 2 and repeat.

For convenience, we refer to completion through Step 6 as the *mutation process*. Additionally, the repeated loop between Steps 2 – 7 (with a killed or stubborn mutant) as one *iteration* of the mutation process. As indicated in Step 7, we stop iterating through the mutation process when there are no more living, non-stubborn mutants.

We now describe Steps 2 and 3 in more detail. We proceeded through the list contained in the "View Mutants and Results" control in MuClipse. The "View Mutants and Results" control in MuClipse displays whether mutants are killed or living. When we inspect a living mutant, we are trying to derive a test case which will detect the change represented by that mutant, which requires a manual code review. Detecting a mutant is a matter of imagining and creating a test case which will reach the line of code, and set up the conditions required to expose the mutant to make its effects visible to the test case. If the mutant's effects cannot be observed, or the line in which the mutation occurred cannot be reached, then we record the mutant as stubborn.

After every iteration, we executed djUnit[10] to indicate which lines were not covered by our final test suites after we attempted to kill all mutants. djUnit is an Eclipse plug-in which executes JUnit tests and creates an HTML report containing line and branch coverage statistics[11]. We updated a listing of the experimentally-determined classification of every mutant as well as the recorded line and branch coverage scores for each iteration through the mutation process.

The order in which mutants are inspected will determine which mutants are labeled as crossfire (see Section 4.1). Although we do not observe a pattern in the order the mutants are presented by this control, it is possible the orderings we used could introduce some statistical bias. The statistical significance of the order of mutant presentation is outside the scope of this work; future studies can further investigate the effect of crossfire mutants.

[10] http://works.dgic.co.jp/djunit/

[11] djUnit does not calculate branch coverage in the conventional way. Consider the following example:

```
        public void method() {
1:          int a = 4;
2:          if (a == 3) {
3:            System.out.println("a == 3");
4:          }
5:        }
```

The branch in the sample is line 2. Depending upon the value of (a==3), the code can proceed to line 3 or line 5. Therefore, there are three lines of code involved in this branch. If the test case involves (a==3) evaluating to false, then lines 2 and 5 are executed and line 3 is not. In this case djUnit computes the branch coverage as 2/3 because 2 of the 3 lines involved in the branch have been executed.

4.4.2. Limitations

Our experimental procedure killed mutants one at a time for two major reasons: 1) the existence of crossfire mutants is invalid if the developer has set out to kill more than one mutant; and 2) finding the greatest number of mutants to be killed with a given test case is not the objective of our study. We declare mutants as crossfire and DOA based on the starting test set and the order in which they are encountered in the list provided by MuClipse. Randomization techniques could have been used to check the experimental significance of the starting test set. For example, one randomization technique would entail running Team A's test cases against Team B's set of resultant mutants. Cross-checking these test cases may prove impossible because a test suite is generally tailored to a specific implementation code and its interfaces. Randomization could also be used on mutant order: killing all mutants in the order in which they appear in MuClipse's mutant view and then killing them again in reverse could provide more experimental information about the crossfire phenomenon. Randomizing mutant order could tell us if crossfire mutants usually die together (regardless of order), or if two mutants' simultaneous deaths are simply a coincidence.

5. Results

In this section, we utilize our research to report results on our research objectives as described in Section 4. Our research questions were as follows:

1) What effect does performing mutation analysis have on the line and branch level coverage of an augmented test suite?
2) Is there a relationship between the mutation operators chosen for mutation analysis and the number of new test cases?
3) Which mutation operators consistently produce the most additional test cases?
4) Does every mutant result in a new test case?
5) Is there any way to evaluate mutation analysis or mutation operators for the objective of augmenting a test suite?

5.1. Mutation Analysis vs. Coverage Analysis

This section discusses the relationship and interaction between performing mutation analysis and coverage analysis's resultant values.

5.1.1. Trends in Coverage and Mutation Score

We recorded line coverage, branch coverage and mutation score to compare progress trends across all projects, classes and teams for each iteration of the mutation process. Figure 5 presents the results of analyzing 360 mutants of the HtmlParser `Attribute` class. Figure 6 presents the results for the 298 mutants of one `Demographics` iTrust class. Each data point represents the classification of a set of mutants after one iteration of the mutation process (see Section 4.1) and an assessment of mutation score and coverage scores. The x-axis represents the number of iterations of the mutation process which have been completed and the y-axis represents the recorded scores (as described in the legend) at the end of the corresponding iteration.

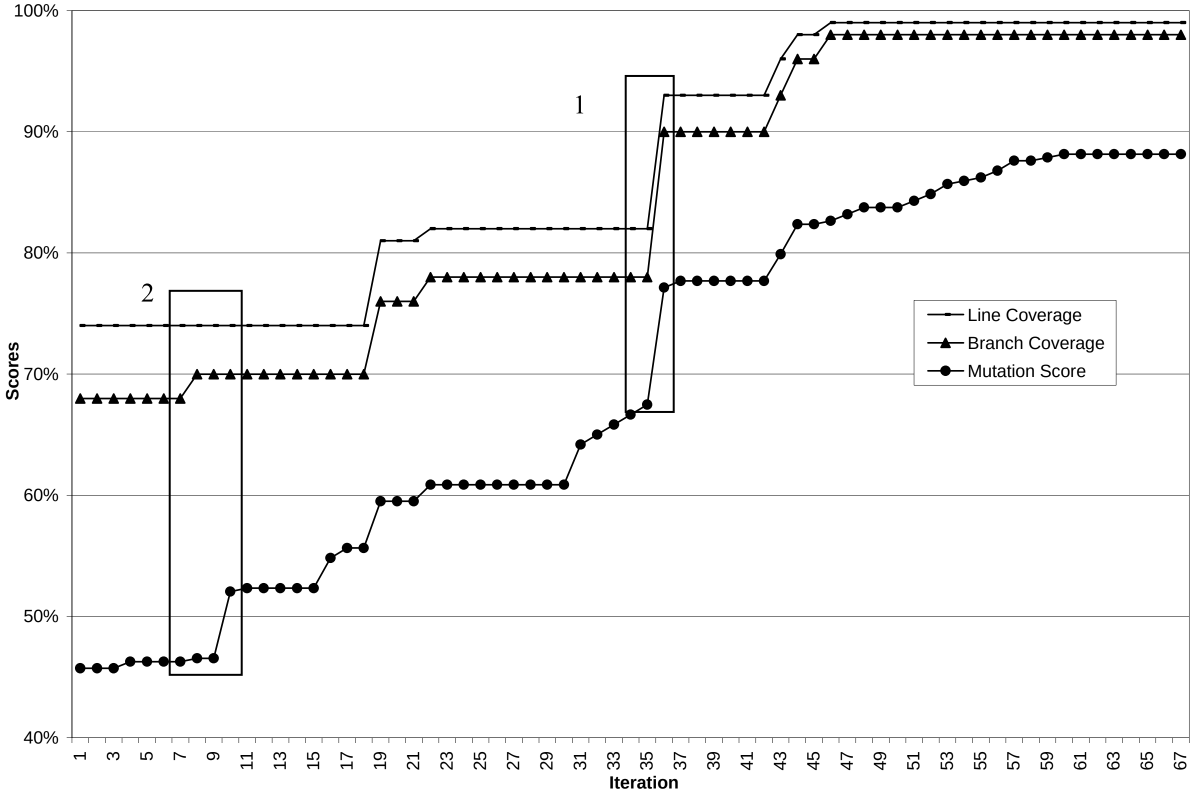


Figure 5. Iteration Progress for 360 mutants of org.htmlparser.Attribute

We chose to provide these examples because they contain well-pronounced instances of several trends observed in the experiment. We enumerate these trends here, ranked in order of benefit to the developer performing mutation testing:

1) When line coverage, branch coverage and mutation score simultaneously trend upward (e.g. Box 1 in Figure 5, iterations 35-36), the developer is killing mutants that are within methods of a given class that are either poorly tested or not tested at all. In these instances, writing a test case to kill a single mutant often provides a large increase in test coverage. Usually these instances are coupled with large increases in mutation score and one or more crossfire mutants.
2) When line coverage stays consistent and branch coverage trends upward with mutation score (e.g. Box 2 in Figure 5, iterations 7-10), the developer is killing mutants that involve segments of conditional control statements. These instances improve branch coverage, which increases the thoroughness of the test cases, but do not involve large jumps in line coverage.

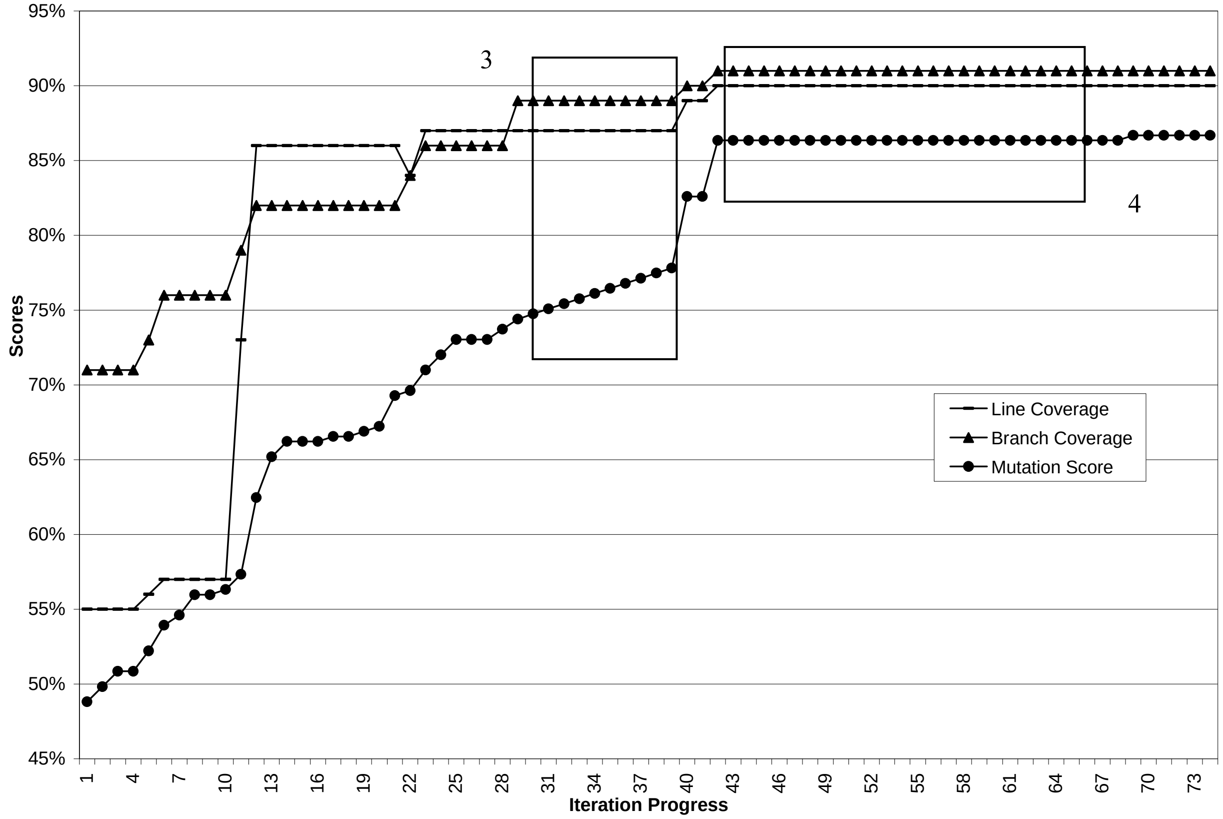


Figure 6. Iteration Progress for 298 mutants of edu.ncsu.csc.itrust.Demographics (Team D)

3) When mutation score trends upward and both line and branch coverage scores remain consistent (e.g. Box 3 in Figure 6, iterations 28-38), the developer is killing mutants that occur in lines of code which have previously been tested but are likely to involve a common programmer error (such as an "off by one error") embodied in a mutation operator. These instances create tests which have fault-detection ability not contained in the existing tests (see Figure 7).
4) When mutation score and coverage scores remain consistent (e.g. Box 4 in Figure 6, iterations 43-67), the developer is trying to kill a series of stubborn mutants. These instances do not provide any new information and consume a significant amount of testing time.

The results for this part of the experiment showed similar trends in every class on which we performed mutation analysis: after completing mutation analysis, the scores in each of line coverage, branch coverage and mutation score had increased. Other Java classes we tested demonstrate slight variations on these trends, but have been left out for brevity.

To illustrate the third trend in the list above, consider the simple method `addIfPositive` presented in Figure 7. `addIfPositive` returns the sum of two numbers if they are positive integers or -1 if either of the two numbers is negative. Assume that initially, the existing test suite does not exercise the method. The original code with an (a,b) input of (1,-1) will return -1, Mutant #1 will return 0 for the same input. Adding this test to the suite increases our line coverage by two lines. However, the input of (1, -1) will return a value of -1 for Mutant #2. Since this return value is the same for Mutant #2 and for the original, Mutant #2 will remain living. To kill Mutant #2, we must use an input of (-1, 1), causing Mutant #2 to return a 0 instead of a -1. This does not cause an increase in line coverage because the line containing the return statement is already covered by the test written to kill Mutant #1. However, it will increase mutation score and strengthen our test suite as in trend three in the list above. Although this case would be eliminated by the boundary value analysis methodology, more complicated examples would not.

These trends occur when performing mutation testing for most classes; however, they do not necessarily occur in the same order or at the same point in the process. For example the large increase in line coverage occurs near iterations 19 and 35 in Figure 5 but nearer to iteration 10 in Figure 6. The occurrence of a trend depends on which mutant is encountered first, which mutants have already been encountered and the pre-existing test suite for a given project.

```
public int addIfPositive
(int a, int b)
{
if (a >= 0 && b >= 0)
   return (a+b);
return -1;
}
```

Original

```
public int addIfPositive
(int a, int b)
{
if (a >= 0 && b < 0)
   return (a+b);
return -1;
}
```

Mutant #1

```
public int addIfPositive
(int a, int b)
{
if (a < 0 && b >= 0)
   return (a+b);
return -1;
}
```

Mutant #2

Figure 7. Method `addIfPositive`

As shown in Table 5, in this study we observed the ratio of average increase in line coverage score to average increase in mutation score is 0.62 upon completion of the mutation process. The ratio of averages can be obtained by dividing the line increase percentage by the mutation increase value in Table 5. Similarly, the average increase in branch coverage score to average increase in mutation score is 0.55. This second ratio of averages can be obtained by dividing the branch increase percentage by the mutation increase value in Table 5. Table 5 contains the

Table 5. Increase in Scores on Average (15 Java classes, average 50 iterations)

	Line	**Branch**	**Mutation**
Start	64%	72%	58%
End	82%	88%	87%
Increase	18%	16%	29%

average across all 15 Java classes in our study for the starting and ending values of line coverage, branch coverage and mutation score. The increase in each value was calculated by subtracting the start value from the end value. These results indicate that completing mutation testing will increase line coverage and branch coverage significantly, but not with every iteration. When coverage is not increasing, mutation testing is driving "common programmer error" testing. None of the measures are at 100%. The mutation operators do not drive test cases to be created for all lines/branches, as will be discussed in the next section. Stubborn mutants prevent a perfect mutation score.

In summary, we find that mutation analysis yields higher branch and line coverage scores by its completion, and there is a direct relationship between line coverage and mutation score. Large increases in line and branch coverage statistics are commonplace in the mutation process and indicate that a reasonable test suite can be augmented using mutation analysis to create a test suite which is more thorough by covering more paths.

5.1.2. Unexecuted Statements

We are interested in how mutation testing will drive the software tester to write new tests to cover all areas of the code for detecting common programmer errors. Therefore, the result of mutation analysis should yield 100% line coverage. Since this value was not reached in any of the 15 Java classes used in our study, we wanted to gather data to categorize the lines of code not executed by any test. After mutation testing was complete for a given class, we executed djUnit to indicate which lines were not executed by our final test suites after we attempted to kill all mutants. Each line of code djUnit marked as not executed was classified into one of the following groups.

- **Body.** Lines that were within the root block of the body of a given method and not within any of the other blocks described below, except for try blocks (see below).
- **If/Else.** Lines that were within any variation of conditional branches (if, else, else if, and nested combinations).
- **Return.** Java `return` statements (usually within a getter or setter).
- **Catch.** Lines that were within the catch section of a try/catch block. Since most of the code in iTrust belongs in a try block (and were therefore counted as body code), only the catch blocks were counted in this category.

Table 6 presents the number of lines which fell into each of the above categories in each project used in our case study. An uncovered line of implementation code usually represents a language construct for which there is no mutation operator. In other instances, the mutants which did exist in these lines were stubborn. In either case, these lines generally reflect weaknesses in the operator set. Most lines that are not executed in the Java classes used in our case study under test fell into a catch section of a `try/catch` block.

Table 6. Quantity of Unexecuted Lines by Team/Project and Class

Team	Class	body	if/else	return	catch
iTrust A	edu.ncsu.csc.itrust.Auth	0	9	3	32
	edu.ncsu.csc.itrust.Demographics	69	50	2	28
	edu.ncsu.csc.itrust.Transactions	0	4	0	11
iTrust B	edu.ncsu.csc.itrust.Auth	5	9	0	28
	edu.ncsu.csc.itrust.Demogaphis	4	27	0	39
	edu.ncsu.csc.itrust.Transactions	0	5	0	11
iTrust C	edu.ncsu.csc.itrust.Auth	0	3	1	33
	edu.ncsu.csc.itrust.Demographics	42	22	0	5
	edu.ncsu.csc.itrust.Transactions	12	1	0	0
iTrust D	edu.ncsu.csc.itrust.Auth	24	4	3	0
	edu.ncsu.csc.itrust.Demographics	25	13	0	2
	edu.ncsu.csc.itrust.Transactions	9	3	0	0
HtmlParser	org.htmlparser.Parser	2	29	3	16
	org.htmlparser.Attribute	0	1	0	0
	org.htmlparser.beans.StringBean	18	2	1	15
Totals		116	182	13	314

Mutation operators for Java exceptions were not included in our operator set. No mutants were created which change, for instance, which Exception is caught, and no test needs to be written to hit the catch block and therefore the catch block remains unexecuted. Research has indicated that killing every non-stubborn mutant will subsume line coverage given an exhaustive set of operators (DeMillo, et al., 1988). However, we find this subsumption only holds when no created mutants are stubborn and when the operator set is exhaustive of all language constructs. Mutation analysis is not going to produce a test to cover every area or possible error which could be found in production code, and the missing areas are going to be in the chosen operator set. Stubborn mutants will not produce test cases, but this fact does not mean that the areas of code where these mutants occur should not be covered by tests.

In summary, we find that while mutation analysis helps drive unit testing for coverage, it will not act as the catch-all solution. We find that the operator set choice is related to the areas of code which will receive the tester's attention, but stubborn mutants also play a role in this determination.

5.2. Operator Analysis

Mutation analysis is computationally expensive and inefficient (DeMillo, et al., 1988). If the process of killing all mutants is to be integrated with a software development methodology, information on how to produce the most test cases and achieve the highest line coverage from mutation analysis is of critical importance. For the purposes of an industrial application of mutation analysis, a test set which achieves the same line coverage score with fewer test cases would be preferable, but since our procedure entails writing a new test case for each mutant, we are interested in the increase in the number of test cases. This section presents empirical information on the behavior of each operator's mutants and whether or not they produce new test cases.

Each iteration classifies some number of mutants into the scheme discussed in Section 4.4. After mutation analysis is complete for a given Java class, totals for each operator were calculated for the number of mutants that were in each classification[12]. The results for each operator are presented in Table 7 and Table 8. The μ values listed in these tables will be explained in Section 5.3.

Table 7. Individual Classification by Operator for iTrust (ranked by μ)

Ops.	Description	Killed	Stbn.	DOA	Crsfr.	Total	μ
AOIU	Insert basic arithmetic ops.	18	1	45	14	**78**	**0.596**
COD	Delete unary conditional ops.	11	0	72	4	**87**	**0.563**
COI	Insert unary conditional ops.	29	5	144	16	**194**	**0.536**
PCI	Type cast operator insertion	5	0	60	13	**78**	**0.532**
AORB	Replace equivalent arithmetic ops.	2	0	21	13	**36**	**0.528**
JDC	Default constructor creation	0	0	4	0	**4**	**0.500**
EMM	Change method modifier	0	0	1	7	**8**	**0.500**
JSD	Delete static modifier	0	0	0	1	**1**	**0.500**
PRV	Replace reference with equivalent	0	0	1	25	**26**	**0.500**
AODU	Delete basic unary arithmetic ops.	0	0	1	0	**1**	**0.500**
AORS	Replace equivalent short-cut arithmetic ops.	0	0	2	0	**2**	**0.500**
ASRS	Replace short-cut assignment ops.	0	0	4	0	**4**	**0.500**
EAM	Change method accessor	62	72	760	247	**1141**	**0.433**
LOI	Insert unary logic ops.	4	7	71	43	**125**	**0.432**
COR	Replace equivalent binary ops.	25	13	54	10	**102**	**0.431**
ROR	Replace relational ops.	20	63	211	67	**361**	**0.266**
JID	Remove variable initialization	0	3	4	0	**7**	**-0.143**
AOIS	Insert short-cut arithmetic ops.	23	109	53	41	**226**	**-0.173**
PCD	Delete type cast ops.	0	35	0	38	**73**	**-0.219**
JSI	Insert static modifier	15	62	2	24	**103**	**-0.330**
OAN	Change Overriding method accessor	0	17	0	0	**17**	**-1.000**
Total		**387**	**214**	**1510**	**563**	**2674**	**0.452**

[12] Unless otherwise stated, "HtmlParser" means the sum for StringBean, Attribute and Parser, and iTrust means the sum for Auth, Demographics and Transactions across teams A-D.

Table 8. Individual Classification by Operator for HtmlParser (ranked by μ)

Ops.	Description	Killed	Stbrn.	DOA	Crsf.	Total	μ
JSI	Insert static modifier	11	0	5	1	**17**	**0.824**
PRV	Replace reference with equivalent	7	0	15	4	**26**	**0.635**
COD	Delete unary conditional ops.	3	0	10	3	**16**	**0.594**
EAM	Change method accessor	10	1	17	20	**48**	**0.573**
ASRS	Replace short-cut assignment ops.	2	0	2	12	**16**	**0.563**
COI	Insert unary conditional ops.	13	1	68	32	**114**	**0.544**
AOIU	Insert basic arithmetic ops.	1	0	7	4	**12**	**0.542**
AORS	Replace equivalent short-cut arithmetic ops.	0	0	2	1	**3**	**0.500**
IOD	Overriding method deletion	0	0	3	1	**4**	**0.500**
JDC	Default constructor creation	0	0	1	0	**1**	**0.500**
COR	Replace equivalent binary ops.	5	2	21	19	**47**	**0.489**
LOI	Insert unary logic ops.	3	3	19	15	**40**	**0.425**
ROR	Replace relational ops.	9	16	77	52	**154**	**0.373**
AORB	Replace equivalent arithmetic ops.	2	2	2	8	**14**	**0.357**
AOIS	Insert short-cut arithmetic ops.	17	24	50	32	**123**	**0.276**
EMM	Change method modifier	3	8	19	3	**33**	**0.182**
IPC	Explicit call to parent's constructor deletion	0	1	0	0	**1**	**-1.000**
JSD	Delete static modifier	0	7	0	0	**7**	**-1.000**
Total		**86**	**65**	**318**	**207**	**676**	**0.419**

The method-level operators AODS, SOR, LOR and LOD did not produce any mutants for either project because they mutated language operators that the chosen classes did not contain (e.g., shift, unary logic, unary arithmetic). Similarly, the class-level operators IHI, IHD, IOP, IOR, ISI, ISD, PNC, PMD, PPD, PCC, OMR, OMD, JTI, JTD, EOA and EOC did not produce any mutants for either project. The operators beginning with 'I' all deal with inheritance features of Java, and the operators beginning with 'O' and 'P' all deal with polymorphism features. The chosen classes were primarily called upon to perform logic checking (in the case of HtmlParser) and interaction with the database (in the case of iTrust), and thus do not employ many of these language features.

Since there is no generalized conclusion to be made from the results at the operator level, grouping operators may yield some new insight. We collected operators into logical groupings based on the "call-letters" of each MuJava operator, as shown in Table 9, with the results shown in Table 10 and Table 11. Some operators, such as Shift and Inheritance were applied to the implementation code, but produced no mutants. These are labeled as `n/a` in Table 10 and Table 11. These results also provide no generalized statement about the relative performance of operators.

The empirical fact that one operator produces more stubborn mutants than another is pertinent, but meaningless until we compare the total number of mutants each operator produces. Inspecting the percentage of each mutant classification produced by each operator includes a measurement of the total number of mutants produced, but this measurement is confounded by the relative meanings of the classifications. As described above, determining the worth of a DOA or crossfire mutant is challenging. Therefore, determining the worth of an operator which produces entirely DOA and crossfire mutants is even more challenging. This problem has motivated us to create a method to summarize the performance of a set of mutants, which is described in the next section.

Table 9. Understanding the MuJava Operators (Ma and Offut, 2006, Ma and Offut, 2005)

Level	Pattern (* = any letter)	Feature (example)
Method	AO**	Arithmetic (+)
	ROR	Relational (==)
	CO*	Conditional (!, &&)
	S**	Shift (<<)
	L**	Logical (&)
	ASRS	Assignment Short-cut (+=)
Class	AMD	Encapsulation
	I**	Inheritance
	P**	Polymorphism
	O**	Overloading
	J**	Java-Specific (e.g. `private` keyword)
	E**	Inter-Object

Table 10. Grouped Classification by Operator for iTrust (ranked by μ)

Feature	Operators Present in Project	Killed	Stbrn.	DOA	Crsf.	Total	μ
Conditional	COD, COI, COR	65	18	270	30	**383**	**0.514**
Shortcut	ASRS	0	0	4	0	**4**	**0.500**
Inter-class	EAM, EMM	62	72	761	254	**1149**	**0.433**
Logical	LOI	4	7	71	43	**125**	**0.432**
Relational	ROR	20	63	211	67	**361**	**0.266**
Polymorph.	PCI, PCD, PRV	5	35	61	76	**177**	**0.218**
Arithmetic	AOIS, AORB, AOIU, AODU, AORS	43	110	122	68	**343**	**0.082**
Shift	n/a	0	0	0	0	**0**	**0.000**
Access	n/a	0	0	0	0	**0**	**0.000**
Inheritence	n/a	0	0	0	0	**0**	**0.000**
Java	JDC, JSD, JSI, JID	15	65	10	25	**115**	**-0.283**
Overloading	OAN	0	17	0	0	**17**	**-1.000**
Total		**214**	**387**	**1510**	**563**	**2674**	**0.323**

Table 11. Grouped Classification by Operator for Project HtmlParser (ranked by µ)

Group	Operators Present in Project	Killed	Stbrn.	DOA	Crsfr.	Total	µ
Polymorph.	PRV,	7	0	15	4	**26**	**0.635**
Short-cut	ASRS,	2	0	2	12	**16**	**0.563**
Conditional	COI, COD, COR,	21	3	99	54	**177**	**0.534**
Logical	LOI,	3	3	19	15	**40**	**0.425**
Inter-class	EMM, EAM,	13	9	36	23	**81**	**0.414**
Relational	ROR,	9	16	77	52	**154**	**0.373**
Arithmetic	AOIU, AOIS, AORB, AORS,	20	26	61	45	**152**	**0.309**
Java	JSI, JSD, JDC,	11	7	6	1	**25**	**0.300**
Inheritance	IOD, IPC,	0	1	3	1	**5**	**0.200**
Shift	n/a	0	0	0	0	**0**	**0.000**
Access	n/a	0	0	0	0	**0**	**0.000**
Overloading	n/a	0	0	0	0	**0**	**0.000**
Total		**65**	**86**	**318**	**207**	**676**	**0.357**

5.3. Operator Utility

In terms of using mutation analysis to drive test set code coverage and produce more test cases, a mutant that produces a new test case (i.e., that is killed) is the most useful. Mutants which are DOA and crossfire are slightly less useful, and a mutant which is stubborn produces no added value and consumes valuable resources. Section 5.3.1 introduces our calculation of utility, which is an aggregate measurement of how often the mutants of a given operator produce new test cases relative to how often the mutant needlessly consumes resources. Section 5.3.2 uses our utility measurements to discuss a diminishing return relationship between the number of new test cases produced by each consecutive mutant.

5.3.1. Definition of Utility

Equation 1 presents our method for ranking the utility (µ) of each mutant operator with respect to the initial data set. µ is a measure of cost (i.e. the amount of time spent dealing with each mutant) versus benefit (i.e. the potential of the mutant to augment a test suite). µ ranges from -1, indicating that the operator produced no useful mutants throughout the sample set, to 1, indicating that the operator produced only killed mutants.

$$\mu = \alpha * \%\,\text{Killed Mutants} + \beta * \%\,\text{DOA Mutants} + \beta * \%\,\text{Crossfire Mutants} - \alpha * \%\,\text{Stubborn Mutants} \qquad \textbf{(1)}$$

We set the coefficients α at 1 and β at .5 to give killed mutants and stubborn mutants a higher impact than DOA and crossfire mutants. Crossfire and DOA mutants can be useful, depending on the order in which the mutants are encountered and the starting test suite, respectively. Because they never take much time, we give them a positive effect. However, since they provide no new test cases, we give them a lesser effect than killed mutants. Stubborn mutants provide potentially infinite cost with no benefit; we give them a negative effect.

Consider the operator ROR, which replaces relational operators (such as ==) with other relational operators (such as !=). In project HtmlParser, ROR produced 154 mutants: 9 killed (5.84%), 16 stubborn (10.38%), 77 DOA (50%) and 52 crossfire (33.76%). Equation 3 gives us μ = 1 * (.058) + 0.5 * (.5) + 0.5 * (.337) – 1 * (.104) = .372. This value indicates that ROR produced useful mutants most of the time for HtmlParser.

The calculated μ values demonstrate several important points regarding our chosen set of mutation operators (see the tables in Section 5.2). First, the average μ for both projects studied is positive, indicating mutation operators which produced mutants for this implementation code have a helpful effect on that code's corresponding test suite overall. Secondly, mutation operators never achieved a μ of 1 in our study, which illustrates that even choosing only those operators which elicit the highest number (proportionately) of new test cases per mutant will generate mutants which do not produce new test cases. Table 12 provides the ranking of these *groups* of chosen mutation operators by their respective μ values. The ranking was different for each project we studied, revealing another important point about our classifications: Aside from a few minor patterns in their relative utilities, mutation operators' utility will change depending on the characteristics of the implementation code.

Table 12. Rankings of Operator Types Across iTrust Teams and HtmlParser

Feature (example)	**iTrust**				**Html Parser**
	A	**B**	**C**	**D**	
Arithmetic (+)	8	6	5	6	7
Relational (==)	4	4	7	5	6
Conditional (`!, &&`)	**1**	**1**	4	**2**	**3**
Shift (<<)	7	5	6	7	10
Logical (&)	5	**3**	**2**	**1**	4
Assignment Short-cut (+=)	7	5	6	3	**2**
Encapsulation	7	5	6	7	10
Inheritance	7	5	6	7	9
Polymorphism	**3**	7	**1**	**3**	**1**
Overloading	7	5	6	9	10
Java-Specific (e.g. static keyword)	**2**	8	8	8	8
Inter-Object	6	**2**	**3**	4	5

Grouping our operators by their MuJava call-letters, we find that conditional, logical and polymorphism operators were in the top three rankings in three out of five projects we studied. For example, the conditional operator COD removed occurrences of the unary conditional operator (!) causing expressions such as `if (! true)` to become `if (true)`. This operator ranked in the top three by μ value in both projects (see Table 7 and Table 8). The unary conditional operator '!' in Java is frequently used to negate a Boolean expression when the desired value is false. In fact, conditional operators were never ranked lower than fourth and had an average ranking across our second experiment (see Table 12). Conditional mutants can quite often be DOA or crossfire, because this error is typically covered using other testing methodologies, such as equivalence class partitioning or boundary value analysis. However, a good number of conditional expressions are overlooked because they have an indirect effect, and thus produce a higher number of killed mutants. Conditional mutants are rarely stubborn because the effect of boolean expressions are typically simple and straightforward to propagate.

Back-end code for most web applications makes use of conditional operators to implement behavior, such as determining the logged in user's authorization for a particular data item. Specifications for a class to perform logic checking and database interaction cause mutation operators effecting conditional operations and arithmetic (AOIU, COI, COD) to be significantly more useful in producing necessary test cases than those operators related to Java class behavior or arithmetic (JSI, PRV, EAM). The object-oriented operators did not provide many useful mutations for iTrust, as the data hierarchy was straightforward. For the framework of HtmlParser, we find that the loosely coupled nature of representing HTML tags causes our class-level operators, such as JSI, PRV and EAM to be more useful (see Table 11). The relationship between mutation operator type and functionality of application reveals that operators should be chosen which are related to the functions the application provides.

In summary, we find that we cannot make a conclusion about which mutation operators are going to produce the highest proportion of new test cases. There seems to be an abstract relationship between which language constructs the system under test employs and which operators are the most effective, but there is no formal or verified method to determine which constructs a system uses.

5.3.2. Operator Utility and Mutation Density

To this end, we have defined mutation density as the number of mutants created within a Java class per line of code as in Equation 2. Table 13 lists the mutation densities for our chosen implementation code from smallest to largest density.

$$\text{Mutation Density} = \frac{\text{\# of Mutants Produced}}{\text{LoC}} \qquad \textbf{(2)}$$

Table 13. Mutation Density and Utility by Java Class (sorted by density)

Class	Total Mutants	Total Lines	Mutation Density	μ (Utility)
Parser	68	318	0.214	0.574
Auth*	552	1135	0.486	0.289
Demographics*	1458	2540	0.574	0.298
StringBean	251	365	0.688	0.410
Transactions*	664	559	1.188	0.406
Attribute	357	263	1.357	0.396
				***Across all iTrust teams**

We now use Equation 1 for the percentage of each mutant in each classification for a given *Java class* (or a group of classes across teams) instead of for a given *operator*. This use of Equation 1 allows us to determine how mutation density in a given class affects the usefulness of mutants produced in that class.

Figure 8 shows the relationship between mutation density and utility rendered from the data in Table 13. The mutation density for `Parser` presents an anomaly which requires further explanation. The Parser class within HtmlParser contained a large number of lines that were in a static method, which the MuJava engine does not mutate. These ignored lines could contain many mutants if the MuJava engine could reach them.

In summary, we find that when using mutation analysis to produce new test cases, the number of mutants produced should not be maximized. Assuming mutants represent actual faults, a test case which finds the first fault will most likely detect the second and third faults which occur in the same area of code. This appears to be related to the Coupling Effect. Just as a test case written to detect a local change will catch the ramifications of this change, the test case will also catch other faults in related areas of code.

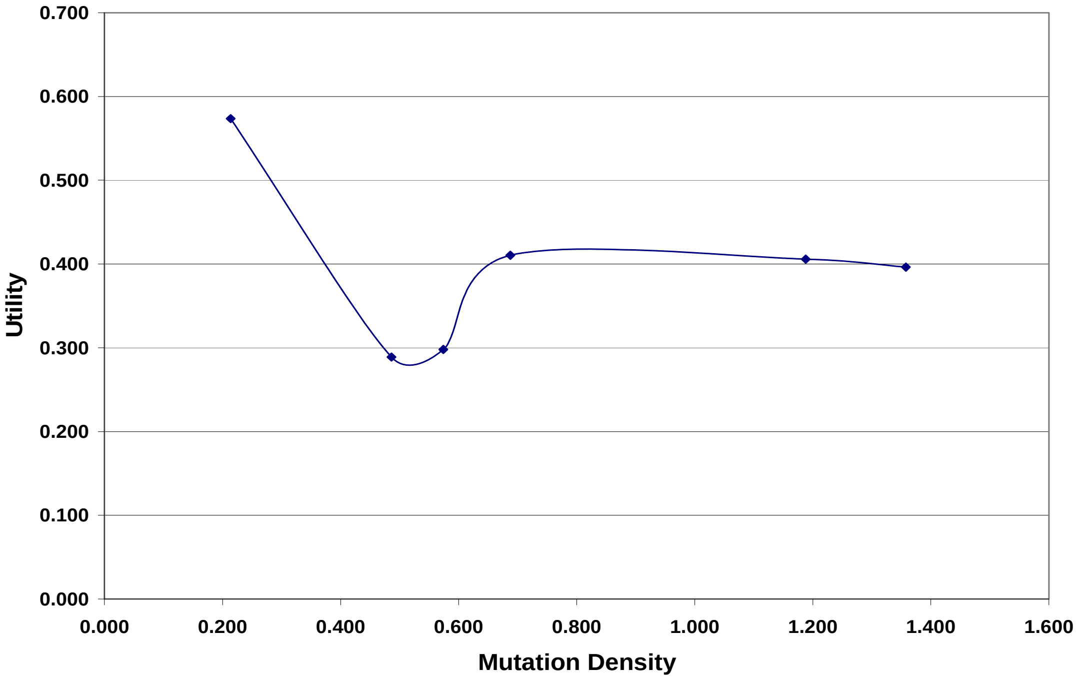


Figure 8. Utility vs. Mutation Density

5.4. Limitations

In this section, we present the limitations of our study.

5.4.1. Study Scope

Our empirical results apply only to the open source projects iTrust and HtmlParser and may not extrapolate to other instances of mutation analysis. iTrust is a relatively large academic code base and larger than the software used in most other mutation testing studies, but still small relative to industrial software. Also, we limited our test set to three classes chosen from each project and team. Testing all of the classes of the iTrust and HtmlParser frameworks would yield more information about the mutation process. Additionally, the set of mutation operators provided by the MuJava framework is more inclusive than some tools by providing object-oriented operators, but its method-level set was chosen based on a selective set determined to sufficiently implement mutation testing (Offutt, et al., 1996). The discarded alternative should be subjected to a similar evaluation of their effectiveness. Additionally, our technique can be used as a verification tool for any newly introduced research mutation operators to evaluate their effectivness.

5.4.2. Determination of Utility

The primary purpose for having the calculation of the utility of an operator is to arbitrarily determine its worth. Future research can incorporate more intricate and complex methodologies for determining the effectiveness of an operator. One suggestion for future work is to optimize the values chosen for the α and β coefficients. Another suggestion would be to investigate the relationship between utility and an objective measurement of initial test suite quality. Finally, a study could include measurements with respect to the severity of faults. We feel the main idea is having a metric for evaluating the behavior of the mutation operators' mutants *as a set*.

While we recognize that line coverage is not considered a perfect metric for determining the adequacy of a test set (Kaner, 1996), the consensus on this topic is that a higher line coverage score yields a less error-prone product (Zhu, et al., 1997). We chose to examine line coverage with respect to mutation score because assuming there is a mutant in every branch or path of the program, killing all the mutants in a program should obtain 100% line coverage. Our study has proven this relationship to not hold due to the existence of stubborn mutants and code which is not mutated (see Section 5.1.2).

6. Summary

Previous research has shown that mutation testing is an effective way to assess the ability of test cases to detect injected faults. We have performed mutation analysis on two open source projects' production code for the following objectives:

- To examine how mutation analysis compares with and/or complements coverage analysis as a means for guiding the augmentation of an automated test suite; and
- To help developers be more efficient in their use of mutation analysis by providing a summary of how many test-creating mutants an operator produces within a moderate sample of source code.

Our results indicate that 1) mutation analysis drives code coverage higher; 2) the operator set used in mutation analysis determines how many new test cases are created and which areas of the production code will receive a tester's attention; 3) there appears to be no conclusive statement to be made about which mutation operators consistently produce the most test cases; 4) mutants which are not equivalent may be somewhat redundant; and 5) mutation operators' behavior and mutation analysis can both be summarized using an aggregate measurement of each derived mutant's ability to create new test cases. Additionally, because of our observations regarding crossfire mutants, we have also concluded that the maximum number of possible mutants produced is not always the optimum for the purposes of increasing line coverage and creating new test cases. In summary, we have shown that mutation analysis is a feasible technique to augment an existing automated test suite.

7. Acknowledgements

We would like to thank the North Carolina State University RealSearch group for their helpful comments on the paper. Funding for this research was provided in part by an IBM Faculty Award. This work is also supported by the National Science Foundation under CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. References

R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia. 2002. Mutation of Java objects. 13th International Symposium on Software Reliability Engineering, Fort Collins, CO USA, 341-351.

J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments? 27th International Conference on Software Engineering, St. Louis, MO, USA, 402-411.

J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. Software Engineering, IEEE Transactions on, Vol. 32, No. 8, 608-624.

R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. 1988. An extended overview of the Mothra software testing environment. Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Alta., Canada, 142-151.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. Computer, Vol. 11, No. 4, 34-41.

P. G. Frankl, S. N. Weiss, and C. Hu. 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. The Journal of Systems & Software, Vol. 38, No. 3, 235-253.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 1995, Design patterns: Elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

R. M. Hierons, M. Harman, and S. Danicic. 1999. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification & Reliability, Vol. 9, No. 4, 233-262.

D. Hovemeyer and W. Pugh. 2004. Finding bugs is easy. ACM SIGPLAN Notices, Vol. 39, No. 12, 92-106.

IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology,"

S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, M. Utting, and R. T. Ltd. 2007. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, 169-175.

C. Kaner. 1996. Software negligence and testing coverage. Proceedings of STAR, Orlando, FL, 1-13.

S. Krishnamurthy and A. Mathur. 1996. On predicting reliability of modules using code coverage. Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, 1-12.

Y. S. Ma, M. J. Harrold, and Y. R. Kwon. 2006. Evaluation of mutation testing for object-oriented programs. 28th International Conference on Software Engineering, Shanghai, China, 869-872.

Y. S. Ma and J. Offut, "Description of class mutation mutation operators for Java," http://ise.gmu.edu/~ofut/mujava/mutopsClass.pdf, *accessed* 4/19/2007.

Y. S. Ma and J. Offut, "Description of method-level mutation operators for Java," http://ise.gmu.edu/~ofut/mujava/mutopsMethod.pdf, *accessed* 4/19/2007.

T. Murnane, K. Reed, T. Assoc, and V. Carlton. 2001. On the effectiveness of mutation analysis as a black box testing technique. Software Engineering Conference, Canberra, ACT, Australia, 12-20.

A. S. Namin and J. H. Andrews. 2006. Finding sufficient mutation operators via variable reduction. Mutation Analysis, 2006. Second Workshop on, 5-5.

J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. 1996. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 2, 99-118.

J. Offutt, Y. S. Ma, and Y. R. Kwon. 2006. The class-level mutants of MuJava. International Workshop on Automation of Software Testing, Shanghai, China 78-84.

J. Offutt, Y. S. Ma, and Y. R. Kwon. 2004. An experimental mutation system for Java. ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, 1-4.

J. Offutt, J. Pan, K. Tewary, and T. Zhang. 1996. An Experimental Evaluation of Data Flow and Mutation Testing. Software Practice and Experience, Vol. 26, No. 2, 165--176.

J. Offutt, J. Pan, K. Tewary, and T. Zhang. 1996. An experimental evaluation of data flow and mutation testing. Software Practice and Experience, Vol. 26, No. 2, 165-176.

B. H. Smith and L. Williams. 2007. An empirical evaluation of the MuJava mutation operators. Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007., Windsor, UK, 193-202.

J. Voas and K. W. Miller. 1995. Using fault-injection to assess software engineering standards. Second IEEE International Software Engineering Standards Symposium 'Experience and Practice', Montreal, Que., Canada, 318-336.

H. Zhu, P. A. V. Hall, and J. H. R. May. 1997. Software unit test coverage and adequacy. ACM Computing Surveys, Vol. 29, No. 4, 366-427.