An Empirical Evaluation of the MuJava Mutation Operators

Ben H. Smith
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
bhsmith3@ncsu.edu

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
williams@csc.ncsu.edu

Abstract

Mutation testing is used to assess the fault-finding effectiveness of a test suite. Information provided by mutation testing can also be used to guide the creation of additional valuable tests and/or to reveal faults in the implementation code. However, concerns about the time efficiency of mutation testing may prohibit its widespread, practical use. We conducted an empirical study using the MuClipse automated mutation testing plug-in for Eclipse on the back end of a small web-based application. The first objective of our study was to categorize the behavior of the mutants generated by selected mutation operators during successive attempts to kill the mutants. The results of this categorization can be used to inform developers in their mutant operator selection to improve the efficiency and effectiveness of their mutation testing. The second outcome of our study identified patterns in the implementation code that remained untested after attempting to kill all mutants.

1. Introduction

Mutation testing is a testing methodology in which two or more program mutations[1] (mutants for short) are executed against the same test suite to evaluate the ability of the test suite to detect these alterations [5]. The mutation testing procedure entails adding or modifying test cases until the test suite is sufficient to detect all mutants [1]. The post-mutation testing, augmented test suite may reveal latent faults and will provide a stronger test suite to detect future errors which might be injected.

The mutation process is computationally expensive and inefficient [3]. Most often, mutation operators[2] produce mutants which demonstrate the need to modify the test bed code or the need for more test cases [3]. However, some mutation operators produce mutants which cannot be detected by a test suite, and the developer must manually determine these are "false positive" mutants. Additionally, the process of adding a new test case will frequently detect more than was intended, which brings into question the necessity of multiple variations of the same mutated statement.

As a result, empirical data about the behavior of the mutants produced by a given mutation operator can help us understand the usefulness of the operator in a given context. Our research objective is to compare the resultant behavior of mutants produced by the set of mutation operators supported in the MuJava[3] tool to empirically determine which are the most effective. Additionally, after completion of the mutation process for a given Java class, we categorized the untested lines of code into exception handling, branch statements, method body and return statements. Finally, our research reveals several design decisions which can be implemented in future automated mutation tools to improve their efficiency for users.

A mutation testing empirical study was conducted using two versions of three major classes for the Java backend of the iTrust[4] web healthcare application. For each Java class, we began by maximizing the

[1] A *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 [5].

[2] A *mutation operator* is a set of instructions for generating mutants of a particular character.
[3] http://ise.gmu.edu/~ofut/mujava/
[4] http://agile.csc.ncsu.edu/iTrust/

efficiency of the existing unit test suite by removing redundant and incorrect tests. Next, the initial mutation score and associated detail by mutant was recorded. We then iteratively attempted to write tests to detect each mutant, one at a time, until every mutant had been examined. Data, such as mutation score and mutant status, was recorded after each iteration. When all mutants had been examined, a line coverage utility was used to ascertain the remaining untested lines of code. These lines of code were then categorized by their language constructs. The study was conducted using the MuClipse[5] mutation testing plug-in for Eclipse. MuClipse was adapted from the MuJava [12] 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 bed 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 details some lessons learned by the gathering of this data which can be applied to the development of future automated mutation tools and which can be used by developers when executing mutation testing in practice.

2. Background and Related Work

Section 2.1 gives required background information on mutation testing. Section 2.2 analyzes several related works on this issue.

2.1 Mutation Testing

The first part of mutation testing is to alter the code under test into several instances, called mutants, and compile them. Mutation generation and compiling can be done automatically, using a mutation engine, or by hand. Each mutant is a copy of the original program with the exception of one atomic change. The atomic change is made based upon a specification embodied in a mutation operator. The use of atomic changes in mutation testing 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 [1]. The Coupling Effect assumes that a test built to catch an atomic change will be adequate to catch the ramifications of this atomic change on the rest of the system [1].

Mutation operators are classified by the language constructs they are created to alter. Traditionally, the scope of operators was limited to the method level [1]. Operators of this type are referred to as traditional or method-level mutants. For example, one traditional mutation operator changes one binary operator (e.g. &&) to another (e.g. ||) to create a fault variant of the program. Recently, class-level operators, or operators that test at the object level, have been developed [1]. Certain class-level operators in the Java programming language, for instance, replace method calls within source 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.

The second part of mutation testing is to record the results of the test suite when it is executed against each mutant. If the test results of a mutant are different than the original's, the mutant is said to be *killed* [1], meaning the test case was adequate to catch the mutation performed. If the test results of a mutant are the same as the original's, then the mutant is said to *live* [1]. *Stubborn* mutants are mutants that cannot be killed due to logical equivalence with the original code or due to language constructs [4]. 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 [10]. However, the inevitability of stubborn mutants may make a mutation score of 100% unachievable. In practice, mutation testing entails creating a test set which will kill all mutants that can be killed (i.e., are not stubborn).

2.2 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." [11]. 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.

[5] http://muclipse.sourceforge.net/

Several researchers have investigated the efficacy of mutation testing. Andrews, et al. [2] chose eight popular C programs to compare hand-seeded faults to those generated by automated mutation engines. The authors found the faults seeded by experienced developers were harder to catch. The authors also found that faults conceived by automated mutant generation were more representative of real world faults, whereas the faults inserted by hand underestimate the efficacy of a test suite by emulating faults that would most likely never happen.

Some researchers have extended the use of mutation testing to include specification analysis. Rather than mutating the source code of a program, specification-based mutation analysis changes the inputs and outputs of a given executable unit. Murnane and Reed [9] illustrate that mutation testing must be verified for efficacy against more traditional black box techniques which employ this technique, such as boundary value and equivalence class partitioning. The authors completed test suites for a data-vetting and a statistical analysis program using equivalence class and boundary value analysis testing techniques. The resulting test cases for these techniques were then compared to the resulting test cases from mutation analysis for equivalent tests and to assess the value of any additional tests that may have been generated. The case study revealed that there was only 14-18% equivalence between the test cases revealed by traditional specification analysis techniques and those generated by mutation analysis. This result indicates that performing mutation testing will reveal many pertinent test cases that traditional specification techniques will not.

Frankl and Weiss [3] compare mutation testing to all-uses testing using a set of common C programs, which contained naturally-occurring faults. All-uses testing entails generating a test suite to cause and expect outcomes from every possible path through the call graph of a given system. 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 find 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. [10] also compare mutation and all-uses testing (in the form of data flow testing), but perform both on the source code rather than its inputs and outputs. Their chosen test bed was a set of ten small (always 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. 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 some results as low as 15%) than that of mutation testing (with the lowest result at 67%).

Ma et al. [6] conducted two case studies to determine whether class-level mutants result in a better test suite. The authors used MuJava to perform mutation testing on BCEL, a popular byte code engineering library, and collected data on the number of mutants produced for both class-level mutation and method-level mutation with operators known to be the most prolific at the latter level. The results revealed that most Java classes will be mutated by at least one class-level operator, indicating that BCEL uses many object-oriented features and that class-level mutation operators are not dependent on each other.

Additionally, Ma et al. completed the mutation process for every traditional mutant generated and ran the resultant test set against the class-level operators. The outcome demonstrated that at least five of the mutation operators (IPC, PNC, OMD, EAM and EMM) resulted in high kill rates (>50%). These high kill rates indicate that these operators may not be useful in the mutation process 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% kill rate, 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.

3. MuClipse

Offut, Ma and Kwon have produced a Java-based mutation tool, MuJava [12], which conducts both automated subtasks of the mutation process in Java 1.4: generating mutants and running tests against created mutants. In generating mutants, MuJava provides both method- and class-level mutation operators. Additionally, developers can choose which operators will be used to generate mutants and where mutant source files would 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 live and killed mutant statistics are displayed at the end of the mutation process.

The MuJava application provided the base for the development of MuClipse, an Eclipse[6] plug-in. Eclipse Runtime Configurations, which run in Java 1.4, are provided for both the generation of and the testing of mutants. In generating mutants, MuClipse allows the developer to decide which mutant operators to employ and which classes should be mutated (see Figure 1). 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 said mutants; and which type of mutant (class- or method-level) operators should be run. MuClipse allows developers the choice of using JUnit[7] 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. MuClipse stores these results as a boolean true, false or Java Exception when collecting original results or mutant results. JUnit is also available in the form of an Eclipse plug in.

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 mutants and their statuses by Java class and mutant type. This View also contains the overall statistics for live and killed mutants, and the calculated mutation score.

Some design decisions were made in the MuJava-to-MuClipse modification to increase the efficiency of MuClipse test execution. Firstly, once a mutant has been killed, MuClipse does not test it again. 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 mutation process. Thirdly, MuClipse does not gather test results for a method within a class that contains no mutants. Finally, so that developers are not required to track this information manually, MuClipse stores live and killed mutant names for a given execution in a file.

4. Research Method

Section 4.1 gives a step-by-step description of the procedure used to conduct our research. Next, Section 4.2 contains information on the design, requirements and architecture of our test bed, the iTrust application. Finally, Section 4.3 details the various terms specific to this study used to classify mutant results throughout the testing procedure.

4.1 Study Procedure

We conducted an empirical evaluation of mutation testing on two versions of the main three classes of the iTrust application. Details on this application will be provided in the next section. For each class, class-level and method-level mutants were generated using all available operators provided by the MuJava framework (which underlies MuClipse).

To characterize the mutation process and gather empirical evidence, the following procedure was followed:

1. Streamline the already-written test suite by removing redundant or meaningless test cases and use the *TestHelper* (described below) class for `setup` and `teardown`.
2. Execute the test cases against all generated mutants (a process provided by MuClipse). Record the classification (e.g. live or killed) of each mutant and record the mutation score. Record all mutants killed by the initial test suite as **DOA**. The full classification scheme will be discussed in Section 4.3.

[6] http://www.eclipse.org
[7] http://www.junit.org/index.htm

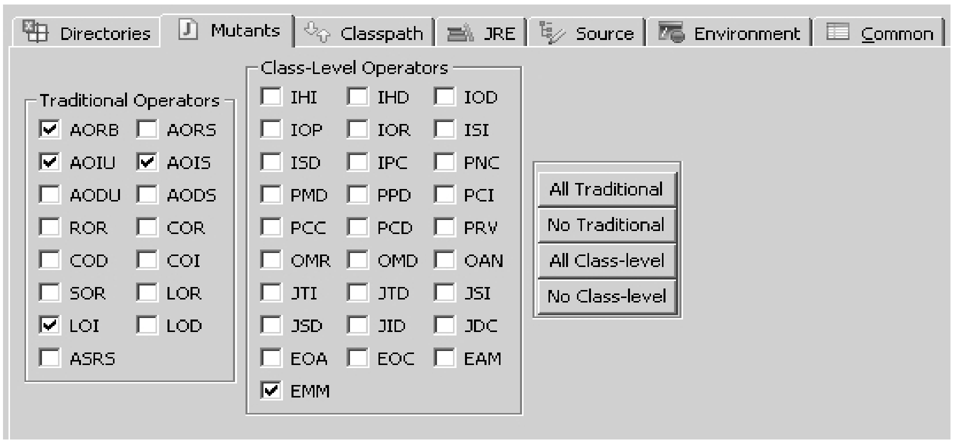
3. Inspect the next (or first) living mutant as it appears in the "View Mutants and Results" control in MuClipse. Attempt to write a test case which will kill this (and only this) mutant.
4. If this mutant can be killed, proceed to Step 5. If this mutant cannot be killed due to language constructs or the need to change the source code, record it as **Stubborn** and return to Step 3.
5. Execute the test cases against the remaining living mutants. Record the mutant in question as **Killed**. Record other mutants that are killed by this test case change as **Crossfire**.
6. If there are no more living, killable mutants, stop. Otherwise, proceed to step 3 and repeat.

iTrust utilizes a database management system to store its data. Since the proper role is required of the logged-in user to execute different functionalities provided by the system, we utilized the *TestHelper* class which inserted users of a desired role and stored their database identifier. In this way, the efficiency of the test suite could be maximized by removing only the users inserted for a given test, as opposed to destroying and restructuring the database with every new scenario.

Additionally, some mutants would cause insertion of faulty data directly into the database while not affecting the return value from a given function or causing an Exception. Mutants of this type had to be checked using database queries from within the test itself.

4.2 iTrust Application

iTrust is a web-based healthcare application that was created by North Carolina State University graduate students in a Software Testing and Reliability course in the Fall of 2005. The best project from the Fall of 2005 was chosen for further enhancement by the same course in the Fall of 2006.



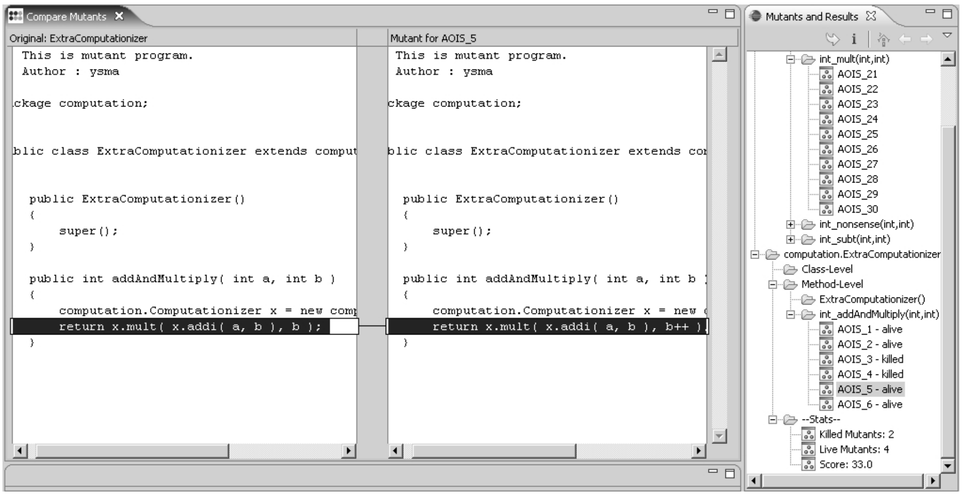
Traditional Operators

[✓ 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



Original: ExtraComputationizer

```
 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 );
  }
```

Mutant for AOIS_5

```
 This is mutant program.
 Author : ysma

ckage computation;


blic class ExtraComputationizer extends co


  public ExtraComputationizer()
  {
      super();
  }

  public int addAndMultiply( int a, int b )
  {
      computation.Computationizer x = new 
      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

The motivation for iTrust was to provide an example project for use in learning the various types of testing and security measures currently available.

iTrust was enhanced by seven two-person teams in the Fall 2006 course. All teams were given the same requirements. We randomly chose two of the seven teams (which we will call Team A and Team B) and conducted mutation testing on three classes of their iTrust framework. We chose the classes of iTrust which performed the computation for the framework and dealt directly with the database management back-end using SQL (Auth, Demographics and Transactions). The other classes in the framework act primarily as data containers and do not perform any computation or logic. Line counts[8] for each Java class in the iTrust framework for Teams A and B are in Table 1. Students were instructed to have a minimum of 80% JUnit statement coverage for each class. We used their JUnit tests as the initial test suite for our empirical study. Statement coverage dropped from 80% to being within a range of 25%-70% when the first step of the procedure was followed.

iTrust was written in Java 1.5 using a Java Server Pages (JSP)[9] front-end. Because the model for the application was to include mostly user interface in the JSP, the three primary classes of the iTrust framework support the logic and processing, including database interaction for the iTrust application. Testing and generation of mutants was executed using Eclipse v3.1 on an IBM Lenovo laptop with a 1.69 Ghz processor and 1.5 GB of RAM running Microsoft Windows XP. Eclipse and MuClipse were executed using Java 1.5 since the test bed source code was written to conform to these standards. 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[10].

4.3 Additional Classification Terms

While classifying mutants traditionally contains the categories **killed**, **living** or **stubborn [4]**, we consider it important not only that a mutant dies, but *when* it dies. A mutant which dies on the first execution of test cases does not yield a new test case, but this might not be true with a different starting test set. Mutants that die with this first execution are called **DOA**. Additionally, 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). Mutants killed by a test case written to kill other mutants are called **Crossfire**.

The MuJava mutation engine (underlying MuClipse) does not operate on compiled binary Java classes, but rather can be thought of as using a regular expression matcher to modify source code using knowledge and logic pertaining to the constructs of the Java language itself. Operating on source code first can lead to two semantically 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

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

//the mutated code
ITrustUser loggedInUser;
```

Figure 3. Logically Equivalent Code

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**.

In sum, we use the following additional terms to classify generated mutants:

- **Killed.** Mutant which was killed by a test case which was specifically written to kill it.
- **Dead on Arrival (DOA).** Mutant that was killed by 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 a test case due to logical equivalence and language constructs.

Killed mutants provide the most useful information: additional, necessary test cases. DOA

[8] LoC calculated using NLOC: http://nloc.sourceforge.net/

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

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

Pkg	Class	Team A	Team B LoC
itrust	***Auth***	**280**	**299**
	AuthenticationException	11	11
	Constants	122	121
	DBManager	171	268
	Demographics	**628**	**544**
	Diagnosis	4	n/a
	Records	285	307
	Transactions	**123**	**120**
	UserDataException	13	14
itrust.bean	DiagnosticInformation	49	49
	Medication	43	43
	OfficeVisit	37	37
	PersonalHealthInformatio	274	560
	TransactionLogRecord	49	49
itrust.users	ITrustUser	77	77
	ITrustAdmin	14	17
	ITrustHCP	14	17
	ITrustPatient	82	83
	ITrustUAP	19	20
	Totals	2295	2636

Table 1. Line Counts by Class and Team

mutants could have provided us with test cases if our initial test suite had been different but might also be considered to be the easiest to kill since they were killed by the initial test suite without any focus by the students on mutation testing. 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. .

5. Results

Mutants were classified to better understand the efficacy of each operator and how their resulting mutants behave. The data collected yield two interesting views of the mutation process. Aggregate classification statistics for mutants of each operator are detailed by Section 5.1. Next, Section 5.2 describes the lines of code marked as not executed by the test suite after completion of the mutation process.

5.1 Classification Statistics

Though some iterations of the mutation process do not kill any mutants, each iteration classifies a number of mutants. After mutation testing is complete for a given Java class, totals for each operator were calculated for the number of mutants that were Crossfire, DOA, Killed, and Stubborn (see Table 2). Descriptions of mutation operators used within the MuJava framework can be found in [7, 8].

Of the 1,330 mutants created in total, almost half (560) were spawned by the operator EAM, with ROR (185) and AOIS (104) coming in second and third respectively. By looking at percentages of mutants created, we can see that the COR, AOIU, COI and COD operators produced the highest number of Killed, or useful mutants. We also find that AOIS, JSI, JID and PCD produced the highest number of Stubborn, or useless mutants. The traditional operators AODU, AODS, SOR, LOR, LOD and ASRS did not produce any mutants for the iTrust back-end because they mutate language operators that the chosen classes did not contain (e.g., shift, unary logic, unary arithmetic). Similarly, the class-level operators AMC, IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC, PNC, PMD, PPD, PCC, OMR, OMD, OAN, JTI, JTD, EOA and EOC did not produce any mutants for the iTrust back-end. 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 and interaction with the database, and thus do not employ many of these language features.

Back-end code for most web applications makes use of conditional operators to implement behavior. Specifications for a class to perform logic checking and database interaction cause mutation operators effecting conditional operations (COR, COI, COD) to be significantly more useful in producing necessary test cases than those operators related to Java classes or arithmetic (JSI, JID, PCD).

The EAM operator replaced one method call to a given class with an equivalent method call, producing a significant anomaly. The coupled nature of the iTrust class structure yielded many mutants by this operator. However, the initial test sets created by both Teams A and B provided simple equivalence class partitioning, which killed these mutants on first execution.

5.2 Unexecuted Statements

After mutation testing was complete for a given class, we executed dJUnit to indicate which lines were not executed by our final test suites. 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, 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 constructor, 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, only the catch blocks were counted in this category.

Most lines not executed in the iTrust classes under test fell into a catch section of a `try/catch` block (see Table 3), corresponding to the fact that mutation operators for Java Exceptions are in development [6]. Since no mutants were created which change, for instance, which Exception is caught, no test needs to be written to hit the catch block and therefore the catch block remains not executed.

A perhaps more disturbing number of lines were found within individual `if` statements and within the body, indicating sections of code which might more likely be executed in a real world setting that were left not executed by the mutation process. The procedure followed precluded fixing errors in the original source code, which caused many of these `if` and `body` instances. When a statement fell after an erroneous line of code, we could not write a test to reach that statement, because we could not fix the error-causing statement before it.

5.3 Limitations

Operator	Description	Killed	Stubborn	DOA	Crossfire	Total
AOIS	Insert short-cut arithmetic ops.	9	62	7	26	**104**
AORB	Replace equivalent arithmetic ops.	2	0	3	7	**12**
COD	Delete unary conditional ops.	6	0	32	2	**40**
COI	Insert unary conditional ops.	17	0	66	10	**93**
COR	Replace equivalent binary ops.	15	6	26	5	**52**
LOI	Insert unary logic ops.	1	7	31	20	**59**
EAM	Change method accessor	20	53	345	142	**560**
JSI	Insert static modifier	3	24	0	24	**51**
JID	Remove variable initialization	0	1	2	0	**3**
PCI	Insert type cast operator	4	0	28	9	**41**
AOIU	Insert basic arithmetic ops.	5	1	16	6	**28**
ROR	Replace relational ops.	10	26	99	50	**185**
PCD	Delete type cast operators	0	35	0	38	**73**
JSD	Delete static modifier	0	0	0	1	**1**
EMM	Change method modifier	0	0	0	2	**2**
PRV	Replace reference with equivalent	0	0	0	25	**25**
JDC	Default constructor creation	0	0	1	0	**1**
Total		**92**	**215**	**557**	**365**	**1330**

Table 2. Classification by Operator (Teams A and B)

Team	Class	catch	if/else	return	body
A	Auth	32	9	3	0
	Demographics	28	50	2	69
	Transactions	11	4	0	0
B	Auth	28	9	0	5
	Demographics	39	27	0	4
	Trans	11	5	0	0
Totals		149	104	5	78

Table 3. Not Executed Lines by Team and Class

Our empirical results apply only to the iTrust application and may not extrapolate to all 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, due to the expensive nature of mutation testing and our iterative mutation killing study, we limited our test set to the three classes listed in Section 4.2. Testing all of the classes of the iTrust framework would yield more information about the mutation process. Additionally, the set of mutation operators provided by the MuJava framework is more inclusive by providing object-oriented operators, but cannot be determined as being representative of every possible operator that could be written. Finally, the expensive nature of the mutation process precludes us from a more real-world procedure of fixing source code that leads to stubborn mutants, which could lead to more insights about the limitations of the mutation process.

6. Lessons Learned

For the Java back-end of a web application we find that conditional operators such as COR, COI and COD provide a substantial number of useful mutations. Back-end code usually performs logical and security checks for a web application and these operators are tailored for this use of code. Additionally, we find that arithmetic operators are not as useful because most functions of this type in a web application are not complicated. Finally, the object-oriented operators did not provide many useful mutations for our web application, as the data hierarchy was straightforward. The correlation between mutation operator type and functionality of application reveals that operators should be chosen which are related to the functions the application provides.

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. 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 source code. Perhaps mutation operators should generate only one or two variations per match.

A lack of exception-driven mutation operators seems to indicate failure in the selected set of operators. However, most of the Exceptions in iTrust deal with errors possibly caused by the database. Error handling is certainly a situation where developers can make mistakes and mutation operators are in development to reflect this fact [6].

New mutation tools can incorporate design decisions to reduce redundant testing (see Section 3) and to automate mutant tracking, but also can include usage of mutation operators which produces fewer instances of a given change and which incorporates Exception mutations.

7. Acknowledgements

Funding for this research was provided by an IBM Faculty Award. Additionally, we would like to thank the North Carolina State University RealSearch group for their helpful comments on the paper.

8. References

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

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," Proceedings of the 27th international conference on Software engineering, pp. 402-411, St. Louis, MO, USA, 2005.

[3] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of

effectiveness," *The Journal of Systems & Software*, vol. 38, no. 3, pp. 235-253, 1997.

[4] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification & Reliability*, vol. 9, no. 4, pp. 233-262, 1999.

[5] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

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

[7] 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.

[8] 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.

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

[10] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience*, vol. 26, no. 2, pp. 165-176, 1996.

[11] J. Offutt, Y. S. Ma, and Y. R. Kwon, "The class-level mutants of MuJava," Proceedings of the 2006 International Workshop on Automation of Software Tesing, pp. 78-84, Shanghai, China 2006.

[12] J. Offutt, Y. S. Ma, and Y. R. Kwon, "An experimental mutation system for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1-4, 2004.