An Analysis of OO Mutation Operators

Jingyu Hu, Nan Li and Jeff Offutt
Software Engineering
George Mason University, Fairfax VA, USA
janetjyhu@gmail.com, nli1@gmu.edu, offutt@gmu.edu

Abstract

This paper presents results from empirical studies using object-oriented, class-level mutation operators. Class mutation operators modify OO programming language features such as inheritance, polymorphism, dynamic binding and encapsulation. Most previous empirical studies of mutation operators used statement-level operators; this study asked questions about the static and dynamic nature of class-level mutation operators. Results include statistics on the various types of mutants, how many are equivalent, new rules for avoiding creation of equivalent mutants, the difficulty of killing individual mutants, and the difficulty of killing mutants from the various operators. The paper draws conclusions about which mutation operators are more or less useful, leading to recommendations about how future OO mutation systems should be built.

1 Introduction

Mutation testing [5] helps testers design high quality tests by making small changes to the program under test, then challenging the tester to design test inputs to make the changes result in failure. The changes are called *mutants* and causing a mutant to result in failure is called *killing* the mutant. A test that kills one or more mutants is called *effective*.

A key to applying mutation to new technologies is the design of effective and efficient mutation operators. Each operator imposes test requirements on the artifact under test, so the quality of mutation testing directly depends on the mutation operators. Most mutation operators for program source mimic typical faults that programmers make, such as using the wrong operator or omitting a statement. Other operators force tests that are usually considered valuable, such as forcing expressions to have the value zero. Still others create uncommon faults, such as changes to a logic predicate that can only be killed by tests that have been found to be effective at finding faults [10], and higher-order mutants make multiple changes to the program [9]. Mutation operators are usually based on empirical or theoretical models of faults [18] and several iterations of experimentation are usually needed to develop and refine a set of mutation operators.

Research into applying mutation to object-oriented (OO) software started about a decade ago [3, 11, 15]. OO testing must check how classes are integrated, which is known as *inter-class testing* [7]. In addition to the traditional issues with method calls, OO software allows class integration to use inheritance, polymorphism, and dynamic binding. The operators that mutate these language features are substantially different from traditional *statement-level* mutation operators, and this paper calls them *class-level* mutation operators, or just *class operators*. Mutants generated from a particular mutation operator are said to be of that mutation *type*.

Class-level mutation operators were implemented in the publicly available muJava tool [16, 17]. Although muJava has been used in many dozens of experimental projects, only two papers have explicitly studied class-level mutation operators. Offutt, Ma and Kwon [21] counted mutants statically, without creating tests or killing mutants. Ma, Harrold and Kwon [14] evaluated OO mutation testing in a study with both static and dynamic components. In the static

study, they used muJava to create 2996 mutants for 266 classes, and presented data on the number of mutants from each mutation operator. Their dynamic study used 11 classes and asked whether tests for traditional mutation could kill object-oriented mutants. They generated tests to kill the traditional, statement-level, mutants, then created 679[1] object-oriented mutants, and then ran the tests on the OO mutants, killing 78%[2] of the OO mutants. Their subjects created mutants for 19 of the 23 mutation operators. Our experimental study has a different goal; we try to evaluate how hard it is to kill mutants created from each operator. It is also larger, as we study 38 classes and 3965 mutants. We also used different versions of muJava, as some of the OO operators were changed as a result of Ma, Harrold and Kwon's study.

Section 2 lists the class-level mutation operators used by muJava. Section 3 presents new conditions under which mutants would be equivalent (*equivalence conditions*). These should be incorporated into future mutation systems to avoid creating equivalent mutants. Section 4 presents static data of mutants for our subject classes, and section 5 presents dynamic data based on tests that kill all non-equivalent mutants in our subject classes. Section 6 discusses these data, draws conclusions, and makes recommendation for future use of mutation testing at the object-oriented testing level. Finally, section 7 concludes the paper.

2 MuJava Class Mutation Operators

Class-level mutation operators were first defined by Kim, Clark and McDermid [11], and Chevalley and Thévenod-Fosse [3, 4]. Offutt et al. developed a categorization of OO programming faults [18], which were used to design a more comprehensive collection of class mutation operators [15, 16, 21]. These mutation operators test the language features of inheritance, polymorphism, dynamic binding, access control and type conversion.

Table 1 gives the 28 mutation operators used by the current version of muJava [17]. The operators are grouped into three categories–Java-specific features inheritance, and polymorphism–based on the language feature that is mutated. The first group includes object-oriented features that are specific to Java. Each mutation operator is denoted by a 3-letter acronym based on a descriptive title. For example, the **i**nheritance operator "Hiding variable Deletion (IHD)" **d**eletes each declaration of a variable that **h**ides a variable of the same name in an ancestor class.

Group	Operator	Description
Java-Specific Features	EAM	**A**cessor **m**ethod change
	EMM	**M**odifier **m**ethod change
	EOA	**R**eference **a**ssignment and content assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	JDC	Java-supported **d**efault constructor **c**reation
	JID	Member variable **i**nitialization **d**eletion
	JSD	`static` modifier **d**eletion
	JSI	`static` modifier **i**nsertion
	JTD	`this` keyword **d**eletion
	JTI	`this` keyword **i**nsertion
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition change
	IOR	**O**verriding method **r**ename
	IPC	Explicit call of a **p**arent's **c**onstructor deletion
	ISD	`super` keyword **d**eletion
	ISI	`super` keyword **i**nsertion
Polymorphism	OAC	**A**rguments of **o**verloading method call **c**hange
	OMD	**O**verloading **m**ethod **d**eletion
	OMR	**O**verloading **m**ethod contents **r**eplace
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PCI	Type **c**ast operator **i**nsertion
	PMD	**M**ember variable **d**eclaration with parent class type
	PNC	**n**ew method call with **c**hild class type
	PPD	**P**arameter variable **d**eclaration with parent class type
	PRV	**R**eference assignment with other comparable **v**ariable

Table 1. Class Mutation Operators

[1]Tables 7 and 8 in their paper give totals of 691, but appear to be in error, as the numbers add up to 679.

[2]Table 8 in give 51%, but the numbers add up to 78%.

3 Equivalence Conditions

A mutant is *equivalent* if it always exhibits the same behavior as the original under all possible test inputs. That is, an equivalent mutant cannot be killed. Detecting equivalent mutants by hand is time-consuming, usually prohibitively so. Two general approaches have been used to deal with the equivalent mutant problem, *detection* and *avoidance*.

Detection efforts have focused on patterns that indicate mutants are equivalent. The resulting algorithms are applied after the mutants are created. Budd and Angluin [2] showed the general problem to be undecidable, so heuristics have been developed to solve the problem partially. Baldwin and Sayward [1] proposed using compiler optimization techniques to detect equivalent mutants. These ideas were developed into algorithms by Offutt and Craft [19] and implemented in Mothra, detecting an average of 10% of the equivalent mutants for 15 program units. DeMillo and Offutt [6] modeled the conditions under which mutants are killed as mathematical constraints that were solved to automatically generate tests. Infeasible constraints in this model represent equivalent mutants. Offutt and Pan [22] developed this idea in an automated tool, which detected an average of 45% of the equivalent mutants and found over 70% of **unreachable statements**. Hierons, Harman and Danicic [8] suggested using program slicing to detect equivalent mutants.

Avoidance efforts started with the 1980-era Mothra system [6]. For each operator, the system defined rules under which a mutant would **not** be created if it would be equivalent [12]. Many of these were used in the original version of muJava, and others were defined later [21].

The following paragraphs define new *equivalence conditions* for class-level operators, that is, conditions under which a mutant would be equivalent if it was created. New equivalence conditions are defined for three mutation operators.

3.1 Java-Specific Mutation Operators

This section defines new equivalence conditions for three Java-specific operators, EAM, JSD, and JSI.

EAM (accessor method change): The EAM replaces an accessor method (*getX()*) with another accessor method. If the original method *getX()* is changed to a mutant *getY()* and method *getY()* only returns *getX()*, the mutant is equivalent and should not be created. All equivalent EAM mutants in this study satisfy this equivalence condition. There is one exception for this equivalence condition, as illustrated by the following example:

Original		**Mutant**

getY ()
{
return getX();
}
	Δ	*getY ()*
		{
		return getY();
		}

The mutant creates an infinite recursive loop, causing a stack overflow exception.

JSD (static modifier deletion): The JSD mutation operator deletes the *static* keyword, thus instead of one copy of a variable being shared among all objects, each object gets a unique copy. If a primitive variable is also defined as *final*, the variable cannot be changed, thus whether it is static does not matter. Thus, the JSD operator should not be applied to *final* variables.

JSI (static modifier insertion): The JSI operator inserts the *static* keyword, thus instead of each object having a unique copy, one copy of a variable is shared among all objects. If a primitive variable is also defined as *final*, the variable cannot be changed, and whether it is static does not matter. Thus, the JSI operator should not be applied to *final* variables.

3.2 Polymorphism Operators

This section defines new equivalence conditions for two polymorphism operators, PCI and PPD.

PCI (Type cast operator insertion): A previous paper [21] gave an equivalence condition for PCI: If the same members are accessed in both original and mutated classes, the mutant is equivalent. This can be expanded in the following way. If an object is cast to one of its ancestor class types, the ancestor class is abstract, and the method to be called is inherited from an ancestor class, the mutant is equivalent. This equivalence condition is illustrated by the following example.

Original:
```
Chronology chrono = getChronology();
if (chrono.getZone() != newZone)
{ ... }
```
Mutant:
```
Chronology chrono = getChronology();
if (((AssembledChronology) chrono).getZone() != newZone)
{ ... }
```

PCI mutants cast object references to their parent types. *Chronology* is a parent class of *AssembledChronology*, which is inherited by class *ISOChronology* and *BuddhistChronology*. *Chronology* and *AssembledChronology* are abstract classes and the method *getZone()* is defined in *AssembledChronology*. The classes *ISOChronology* and *BuddhistChronology* are final classes that inherit *getZone()*. Therefore, when the object *chrono* is an instance of *ISOChronology* or *BuddhistChronology*, *chrono.getZone()* returns the same result no matter whether *chrono* is cast to its parent class type. Thus, *((AssembledChronology) chrono).getZone ()* and *chrono.getZone ()* return the same result and the mutant is equivalent. PCI should not be applied if the method called is defined in the parent class and is not overridden by any child class.

PPD (Parameter variable declaration with parent class type): PPD changes the type of a formal parameter to be its parent type. Class *B* inherits from class *A*, both *A* and *B* implement an interface that declares *getY()*, and *getY()* is implemented in *B*, but not *A*. Therefore, if an object of type *B* is cast to type *A*, *getY()* in *B* is still called, then casting *B* to *A* does not change which methods are called. Thus the mutant would be equivalent.

Original:
```
getX (ClassB instanceofB)
{
    instanceofB.getY();
}
```
Mutant:
```
getX (ClassB _instanceofB)
{
    ClassA instanceofB = (ClassA) _instanceofB;
    instanceofB.getY();
}
```

In the example, class *A* is the parent class of class *B*. Method *getY()* is inherited from *A* and not overridden by *B*. Therefore, the mutant returns the same result as the original program and the mutant is equivalent.

3.3 Equivalence Conditions Summary

A hand analysis of the classes used in this study found 109 EAM, 48 JSD, 260 PCI, and 2 PPD equivalent mutants (table 2). The equivalence conditions in this section would have found all the EAM and JSD equivalent mutants, 111 of the PCI equivalent mutants. and 1 of the PPD equivalent mutants.

4 Counting Class Mutants

A major factor in the cost of any test criterion is the number of test requirements and the number of tests needed to satisfy those requirements. For example, statement coverage has one test requirement for each statement in the program unit under test, but far fewer tests are usually needed to cover all the statements. Statement-level mutation is known to generate more test requirements (that is, mutants) than other test criteria, but a recent study found that it does not necessarily need more tests than criteria such as all-uses and prime paths [13].

This section quantifies information about mutants on 38 Java classes drawn from freely available programs: *CommissionEmployee* and *Employee* from Deitel's Java text, *Point* from a homework assignment at GMU, and open source programs *org.joda.time* (*http://joda-time.sourceforge.net/*) and *javassist* (*http://www.jboss.org/javassist/*). We used all classes from *Point*, *CommissionEmployee* and *Employee*, and classes in the presentation layer of *org.joda.time* plus two others that allowed us to create JID and PCC mutants.

4.1 Classes Studied

Figure 1 summarizes descriptive statistics from the classes. Each stacked bar represents a different statistic, with data split into groups depending on the statistic. The number of lines (first bar) is divided into classes with fewer than 100, 101 to 200, 201 to 500, and more than 500 lines. 32% of the classes have fewer than 100 lines of code and over 58% have more than 100 and fewer than 500. Most classes have fewer than 500 LOC, although 10% have over 500. The inheritance depth varies from zero to three.

Almost half the classes have only one constructor, but 26% have more than 10. The other bars show

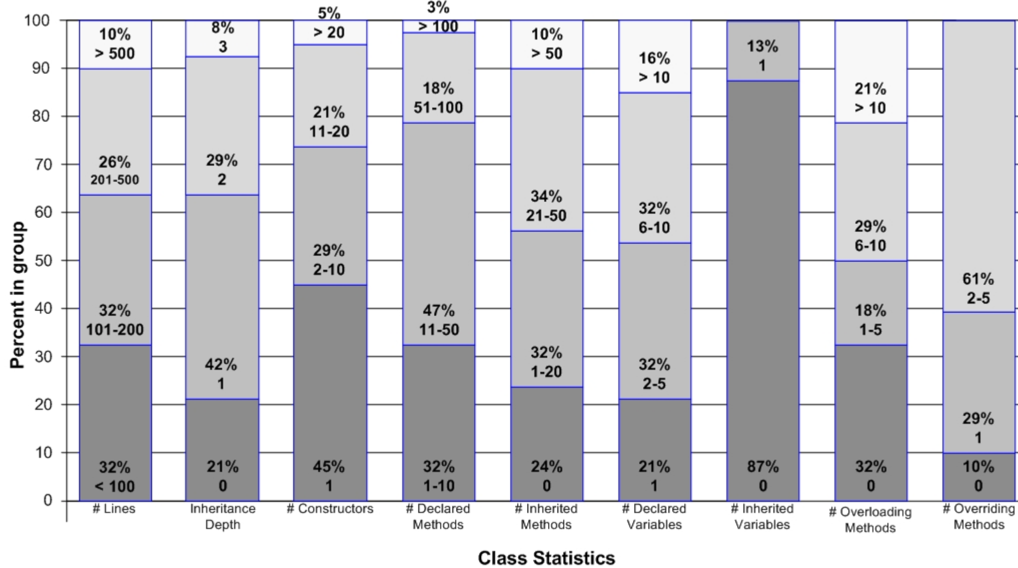

Figure 1. Statistics from the 38 Classes

the number of methods declared in classes, number of methods inherited, number of class-level (instance) variables declared and inherited, and the number of overloading and overriding methods.

The inheritance depth does not include Java default classes (including *java.lang.Object* and *java.lang.Exception*). Declared methods do not include constructor or overriding methods. If the methods *equals (Object o)*, *int hashCode()*, or *String toString()* are overridden in a class and not in an ancestor class, they are considered to be overriding, not declared methods. The overloading methods in a class do not include methods inherited from ancestor classes. The overriding methods in a class do not include methods inherited from Java default classes.

4.2 Mutants and Equivalent Mutants

Table 2 shows the number of mutants that were created by each mutation operator. The first column is the mutation operator. The second and third columns show the number of mutants created by each operator and the number of equivalent mutants from this study. Equivalence was determined by hand with the agreement of both the first and second author. The last three columns show the percentage of all mutants that were equivalent, the percentage of all mutants that were created by each operator, and the percentage of all equivalent mutants that were created by each operator.

Three operators, EAM, OAC and PCI, accounted for 87% of all mutants in the classes studied.

4.3 Number of Mutants Per Class

The number of mutants per class varies greatly. MuJava does not mutate abstract classes, so they were not used in this study. But the real differences arise from the fact that class-level mutants depend on specific language features, many of which are used in some classes but not others, and some of which are rare.

Figure 2 summarizes the number of mutants per class by dividing the classes into five groups. The largest group, 52.6%, had between 11 and 50 mutants. Only one class had more than 1000 mutants, and it had 1277. That class had 840 OAC mutants, which accounted for 24.9% of all the mutants.

5 Killing Class Mutants

For the next part of the study, we designed a total of 575 tests for the 38 classes, killing 3398 mutants. We analyzed the live mutants by hand and determined that 492 were equivalent, leaving only 75 that we were not able to kill. The overall mutation score was .98.

This section reports results from this effort, which is the most comprehensive study reported on class-level

Operator	# Mutants	# Equiv	% Equiv	% All Muts	% All Equiv
EAM	1066	109	10.2%	26.9%	22.4%
EMM	129	0	0.0%	3.3%	0.0%
EOA	3	0	0.0%	0.1%	0.0%
EOC	7	7	100.0%	0.2%	1.4%
JDC	1	0	0.0%	0.0%	0.0%
JID	3	0	0.0%	0.1%	0.0%
JSD	53	48	90.6%	1.3%	9.9%
JSI	97	0	0.0%	2.4%	0.0%
JTD	8	0	0.0%	0.2%	0.0%
JTI	9	1	11.1%	0.2%	0.2%
IHD	1	0	0.0%	0.0%	0.0%
IHI	15	0	0.0%	0.4%	0.0%
IOD	75	32	42.7%	1.9%	6.6%
IOP	6	2	33.3%	0.2%	0.4%
IOR	12	5	41.7%	0.3%	1.0%
IPC	59	0	0.0%	1.5%	0.0%
ISD	8	0	0.0%	0.2%	0.0%
ISI	45	11	24.4%	1.1%	2.3%
OAC	934	5	0.5%	23.6%	1.0%
OMD	1	0	0.0%	0.0%	0.0%
OMR	55	1	1.8%	1.4%	0.2%
PCC	8	0	0.0%	0.2%	0.0%
PCD	7	4	57.1%	0.2%	0.8%
PCI	1282	260	20.3%	32.3%	53.4%
PMD	3	0	0.0%	0.1%	0.0%
PNC	19	0	0.0%	0.5%	0.0%
PPD	2	2	100.0%	0.1%	0.4%
PRV	57	0	0.0%	1.4%	0.0%
Total	**3965**	**487**	**12.3%**		
Average (*per class*)	**104**	**13**			

Table 2. Number of Class Mutants

mutants. We first designed tests to kill the mutants, then eliminated duplicate (ineffective) tests. We then ran each test against each mutant (not just the live mutants, as in normal mutation usage). This allows individual mutants and mutation operators to be compared on the basis of strength.

5.1 Class-Level Mutant Integration Tests

Unit tests and intra-class tests are usually short and simple; the test calls a method with specific values. Sometimes other methods must be called first to put needed values into an object or shared instance variables. Tests to kill class-level mutants can be more complicated because the integration nature of the mutants means we need inter-class tests. For some mutants, a test to kill it includes a new child class that inherits from the mutated class, then a sequence of calls to methods in the new class. This subsection includes two example tests to give an idea of how this kind of integration test design differs from unit testing.

In the first example, taken from *Period* in *org.joda.time*, the PMD operator changed the member variable *ZERO* from type *Period* to *Period*'s parent class, *BasePeriod*. Thus, a reference to *Period.ZERO* causes a *NoSuchFieldError* exception.

```
public String test1()
{
    Period period = Period.ZERO;
    return period.toString();
}
```

In the second example, taken from *MutableDateTime* in *org.joda.time*, the ISI operator changed a call to a method in a child class to use the parent's version, by inserting the *super* keyword. The child class *MutableDateTime* inherits from its parent class *DateTime*. *BaseDateTime* defines a method *setMillis()*,

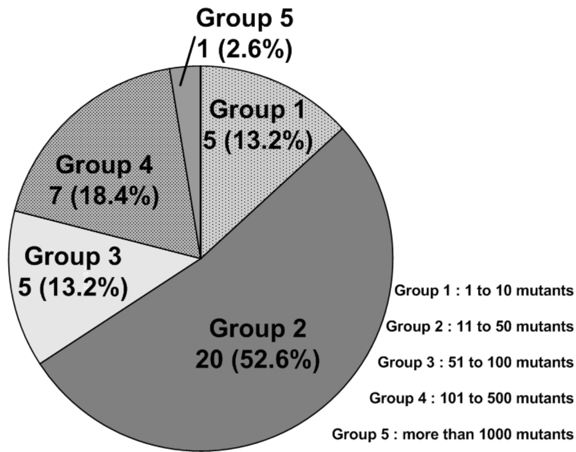

Figure 2. Number of Mutants Per Class, Grouped in Ranges

which does not round the value. *MutableDateTime* overrides *setMillis()* by adding rounding. *setMillis()* is called from the method *add()* and is mutated to be *super.setMillis()*. To kill the mutant, the test must call *add()*, after creating a *MutableDateTime* object and calling *setRounding()* on that object. The parameter values must also be chosen such that the difference in rounding will be shown.

```
public String test71()
{
    MutableDateTime mdt = new MutableDateTime
        (ISOChronology.getInstanceUTC().getDateTimeMillis
        (2000, 10, 02, 13, 34, 56, 789));
    mdt.setRounding (BuddhistChronology.getInstanceUTC().
        hourOfDay(), 2);
    DateTime start = new DateTime (2004, 12, 25, 0, 0, 0, 0);
    DateTime end = new DateTime (2004, 12, 25, 5, 6, 7, 0);
    Duration dur = new Duration (start, end);
    mdt.add (dur);
    return mdt.toString();
}
```

5.2 Number of Tests That Killed Each Mutant

Figure 3 summarizes how many mutants were killed by the tests, grouped in percentages. The first bar shows that 75 mutants were **not** killed (the chart does not show 492 equivalent mutants). The second bar shows that 828 mutants were killed by at least one test, but no more than 5% of the tests. The peak is in the third bar; 1432 mutants were killed by between 6% and 10% of the tests. After that, the numbers fall off rapidly. Relatively few mutants were killed by more than 10% of the tests. These percentages were counted separately for each class, then aggregated across the 38 classes. That is, we counted how many mutants were killed by 5% or fewer of the tests for each class, then summed across the classes. This was necessary because the tests for one class cannot kill mutants from other classes. This graph is very different from what is usually seen for statement-level mutants, where most mutants are killed by a large majority of the tests.

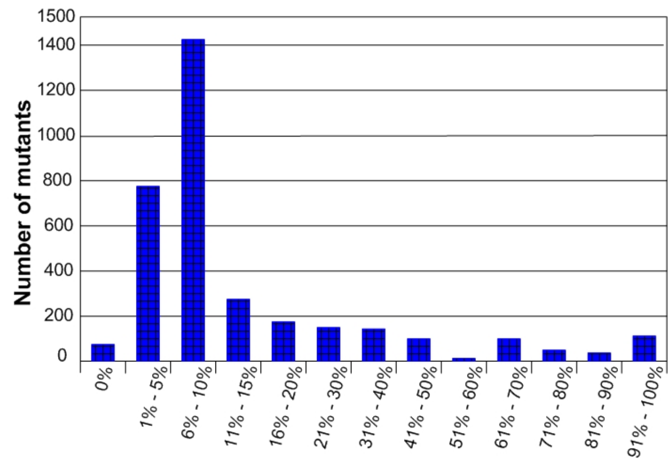

Figure 3. Number of Mutants Killed, Grouped by Percentages

5.3 Estimating Mutation Operator Strength

Running every mutant in a class against every test for a class provides data that can be used to estimate the strength of each operator. We consider an operator to be "strong" if mutants generated for that operator are killed by relatively few tests; an operator is "weak" if its mutants are killed by lots of tests.

Equation 1 defines *Mutation Operator Strength* (*MOS*). The numerator, *#minTests*, is the minimal number of tests needed to kill all killable mutants of that type in that class. As usual, a test set is *minimal* if removing any test cases would cause the test set to no longer be mutation adequate. As an example, we generated 19 tests for the class *IllegalFieldValueExcep-*

tion. Eleven of those were needed to kill 17 of the 19 mutants of type reference assignment with other comparable variable (PRV). So for class *IllegalFieldValueException* on mutation operator PRV, *#minTests = 11*. The denominator is the number of mutants killed. The ratio is the number of tests needed to kill the mutants over the number of mutants. The tests are minimal, so the size of the test set is bounded by the number of mutants. If a single unique test is needed to kill each mutant, the ratio will be 1. If one test can kill all mutants, the ratio is small. Thus, this formula gives a rough approximation of the strength of an operator; if a mutation operator creates stronger mutants, the ratio is higher.

$$MOS = \frac{\#minTests}{\#muts - \#notKilled} \quad (1)$$

We killed no mutants for two of the 28 class-level operators in muJava. All seven EOC and all two PPD mutants were equivalent.

Figure 4 summarizes the MOS measure on the data on the 38 classes. Only the 26 mutation operators for which we were able to kill mutants are included. Interestingly, most of the operators should be considered very strong. Twelve have a strength of 1, 18 have strength of .8 or above, 21 above .5, and only five have strength below .2. PRV's strength is .509, low but not as low as EAM, OAC, PCC, PCI and PNC. The Mothra selective research [20] found that over half the statement-level operators were relatively weak; well under .5 by the measure defined in this paper. So data on class-level mutation operators are quite different.

6 Findings and Recommendations

This section discusses specific findings from this study and makes recommendations about the use of class-level mutation operators.

6.1 General Findings

The classes studied averaged 104 class-level mutants per class, although with lots of variation. 12.3% of the mutants were equivalent, which is consistent with statement-level mutants. However, if the equivalence conditions in section 3 were applied, the percentage of equivalent mutants on these classes would be only 5.9%.

An interesting finding is how different class-level mutation is from statement-level mutation. Most statement-level mutants are killed by far more tests than most class-level mutants. The number of tests needed to achieve near-100% mutation score on class-level mutants was 14.5% the number of mutants (a ratio of 6.89 mutants per test). We usually need fewer than 5% of the number of tests to kill all statement-level mutants [20]. A positive interpretation of this is that we have less overlap among class-level mutation operators. Unfortunately, we are unlikely to see a significant decrease in the number of mutation operators needed, as we did with statement-level mutation.

Generally speaking, we found that class-level mutation operators target relatively subtle faults that are related to how classes are coupled. Many of these faults are probably fairly rare, but potentially very expensive to find and fix.

6.2 Findings Based on Operator Strength

The Arguments of Overloading Method Call Change (OAC) operator only had a strength of .011. In addition, the classes studied had 934 OAC mutants, for 23.6% of all mutants. Thus the benefit of OAC seems really low. These findings lead us to recommend that OAC **not** be included in future OO mutation systems.

Similarly, the Type Cast Operator Insertion (PCI) operator only had a strength of .079. In addition, the classes studied had 1282 PCI mutants, for 32.3% of all mutants. These findings indicate that the PCI operator is not only unhelpful, but also expensive. Thus we recommend that PCI **not** be included in future OO mutation systems.

The next finding is about the Accessor Method Change (EAM) and Modifier Method Change (EMM) operators. Tests that killed all EAM mutants also killed most EMM mutants, and conversely, tests that killed all EMM mutants killed most EAM mutants. However, the data are not strong enough to justify eliminating one of the two operators in response to this apparent redundancy. EAM and EMM modify different methods and some classes have mutants of one type but not the other. Rather, we suggest a more nuanced approach. If a class has mutants from one of, but not both, EAM and EMM, the mutation engine

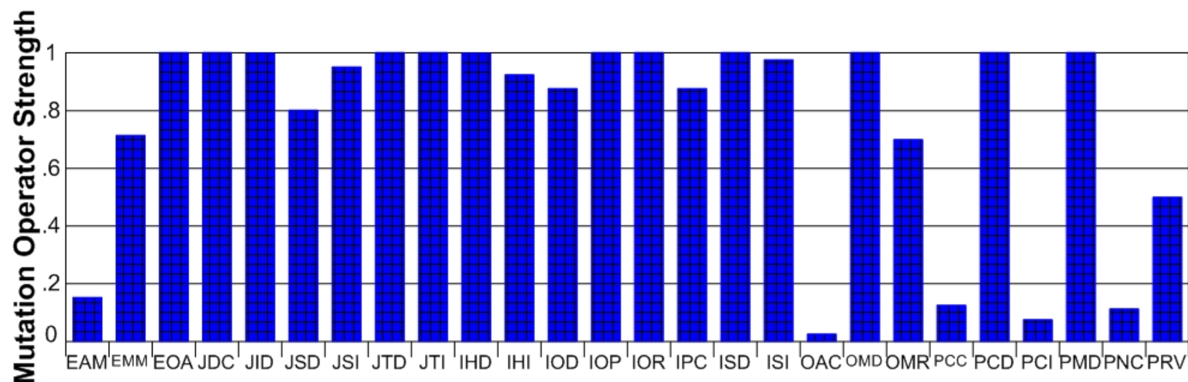

Figure 4. Strength of Mutation Operators

should create those mutants. But if a class has mutants from both, only one operator should be used. It may not matter which, but we recommend using the operator that generates the fewest mutants. Since EAM accounted for 26.9% of all the mutants, and 22.4% of all the equivalent mutants, this approach could yield great savings.

Taken together, these three findings have the potential to eliminate 82.8% of the mutants in the 38 classes studied!

We also found that all seven EOC mutants were equivalent, leading us to question whether the operator is useful. Unfortunately, we were not able to define equivalence conditions for this operator.

6.3 Threats to Validity

An external threat to validity is that we looked at 38 classes, and there is no guarantee that the results on those classes will hold on all classes. We chose the subject classes from real software to be as representative as possible. Several potential internal threats to validity exist. We only generated one test suite per class and it is possible that different test suites would yield different results. However, we have never seen a study that used multiple test suites where the additional test suites actually made a difference. Equivalence and expected output was determined by hand; to eliminate bias, two people verified each kill and equivalent mutant determination. Mutant creation was managed by the tool muJava, so any flaws in its execution could impact the results. Finally, it is possible to design many small tests or fewer large tests. We factored this possible affect out by using percentages in table 3.

7 Conclusions

This study is the most comprehensive study of killing class-level mutants. In addition to providing hard data on classes and class-level mutants, this study looked at several dynamic aspects of class-level mutants. The study carefully counted equivalent mutants and this paper provides the first data on how many equivalent class-level mutants can be expected. The study also gathered detailed data on killing mutants. We generated 575 tests for 38 classes, killing 98% of the non-equivalent mutants (3398).

Normal use of mutation runs new tests only against mutants that have not been killed by previous tests. This study ran **every** test against **every** mutant to measure how hard it is to kill individual mutants, and mutants from specific operators. This analysis led to several findings, as detailed in section 6. Specifically, we recommend eliminating the class-level mutation operators OAC and PCI, and only using one of the operators EAM and EMM. We also identified equivalence conditions for the mutation operators EAM, JSD, JSI, PCI and PDD. These conditions should be included in future mutation generators. For the classes studied, they would have eliminated 268 equivalent mutants, bringing the percentage of equivalent mutants down from 12.3% to 5.9%.

A careful look at table 2 and figure 4 indicates a possible inverse correlation between the number of mutants generated by an operator and its strength, but we are not convinced this is generally true. And if it was,

we are not sure why. It is possible that operators that produce more mutants create more overlap among the mutants.

Taken together, the findings of this study can make future OO mutation systems more effective and significantly more efficient.

References

[1] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Research report 276, Department of Computer Science, Yale University, 1979.

[2] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[3] P. Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macau SAR, China, December 2001.

[4] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, September 2002.

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

[6] R. A. DeMillo and J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[7] L. Gallagher, J. Offutt, and T. Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability, Wiley*, 17(1):215–266, January 2007.

[8] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability, Wiley*, 9(4):233–262, December 1999.

[9] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, To appear, 2010. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[10] G. Kaminski and P. Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009)*, pages 356–365, Denver, CO, April 2009.

[11] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented strategies with the mutation method. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 4–100, San Jose, CA, October 2000.

[12] K. N. King and J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[13] N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Fifth Workshop on Mutation Analysis (Mutation 2009)*, Denver CO, April 2009.

[14] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon. Evaluation of mutation testing for object-oriented programs. In *Proceedings of the 28th International Conference on Software Engineering*, pages 869–872, Shanghai, China, May 2006. IEEE Computer Society Press.

[15] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis MD, November 2002. IEEE Computer Society Press.

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

[17] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. muJava home page. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/.

[18] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong China, November 2001. IEEE Computer Society Press.

[19] J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification, and Reliability, Wiley*, 4(3):131–154, September 1994.

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

[21] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of muJava. In *Workshop on Automation of Software Test (AST 2006)*, pages 78–84, Shanghai, China, May 2006.

[22] J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability, Wiley*, 7(3):165–192, September 1997.