The Class-Level Mutants of MuJava

Jeff Offutt
Info. & Software Eng.
George Mason University
USA
offutt@ise.gmu.edu

Yu-Seung Ma
Embedded Software Research
Elec. & Telecom. Rsrch Inst.
Korea
ysma@etri.re.kr

Yong-Rae Kwon
Computer Science
EE & CS
KAIST, Korea
kwon@salmosa.kaist.ac.kr

ABSTRACT

This paper presents results from empirical studies of object-oriented, class level mutation operators, using the automated analysis and testing tool MuJava. Class mutation operators modify OO programming language features such as inheritance, polymorphism, dynamic binding and encapsulation. This paper presents data from 866 classes in six open-source programs. Several new class-level mutation operators are defined in this paper and an analysis of the number of mutants generated is provided. Techniques for eliminating some equivalent mutants are described and data from an automated tool are provided. One important result is that class-level mutation operators yield far more equivalent mutants than traditional, statement-level, operators. Another is that there are far fewer class-level mutants than statement-level mutants. Together, these data suggest that mutation for inter-class testing can be practically affordable.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification

1. INTRODUCTION

Mutation testing [7] assumes that a program will be well tested if most simple faults are detected and removed. Simple faults are introduced by creating a set of faulty versions, called *mutants*. *Mutation operators* describe syntactic changes to the programming language and are applied to the original program to create mutants. Test cases are created with the goal of causing each mutant to produce incorrect output. A test case that causes one or more mutants to fail is called *effective*.

Mutation testing has a rich history with advances in concepts, theory, technology, and empirical evidence [1, 9, 19]. 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. A good set of mutation operators is crucial to the success of mutation testing, and different operators are needed for each language and different types of testing. Mutation operators follow two general strategies; modeling realistic faults, and forcing testers to apply common test heuristics.

Although object-oriented (OO) programming languages have been in use since the early 1990s, research on mutation testing of OO languages only started recently. Most of the research thus far has focused on the first two activities. Mutation operators [12, 5, 13] for *inter-class testing* [10] have been developed to focus on OO language features such as inheritance, polymorphism, and dynamic binding. These operators are substantially different from traditional *statement-level* mutation operators. The most recent tool is the publicly available Java mutation tool MuJava [14, 15].

This paper presents enhancements to the MuJava tool (new mutation operators and an equivalence detection engine) and data from applying class mutation operators to several open source software programs. The new mutation operators are associated with Java type conversion.

2. NEW CLASS MUTATION OPERATORS

MuJava automatically generates mutants, runs the mutants against tests supplied by the tester, and reports the mutation score of the test suite. MuJava can be obtained from the two mirrored websites at KAIST and GMU [15]. MuJava has been downloaded many dozens of times and we are currently aware of classroom and research use at eight universities.

Class mutation operators were first proposed in papers by Kim, Clark and McDermind [12], and Chevalley and Thévenod-Fosse [5, 6]. Offutt et al. developed a categorization of OO programming faults [20], which we used to design a more comprehensive collection of class mutation operators [13, 14]. There are two goals for these mutation operators, first to address all the OO programming faults and second to ensure all OO language features are tested. These mutation operators tesed the language features of inheritance, polymorphism, dynamic binding, and access control. This paper introduces operators to test type conversion features, a crucial capability in OO languages. These operators are discussed in the following subsection..

In addition to new mutation operators, some existing operators have been refined for MuJava version 2. The new

AST'06, May 23, 2006, Shanghai, China.

version contains three new mutation operators for type conversion, merges two operators from the old version into one, and splits three different operators into two apiece (three operators became six). These changes make the operator definitions and implementations more consistent. Thus, MuJava Version 1 had 24 mutation operators and Version 2 has 29. Table 1 lists MuJava's current set of mutation operators. Comparing with Table 1 in our previous paper [14], Table 1 in this paper splits ISD into ISD and ISI, splits JSC into JSI and JSD, splits JTD into JTD and JTI, merges OAO and OAN into OAC, and introduces three operators related to type conversion, PCI, PCC, and PCD.

Each mutation operator is denoted by a 3-letter acronym based on a descriptive title. For example, the "Access Modifier Change (AMC)" operator replaces each access control modifier in a program by each other modifier.

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

Table 1: Class Mutation Operators

2.1 Type Conversion Mutation Operators

Dynamic binding allows object references to have different types during execution. Object references may refer to objects whose **actual types** differ from their **declared types**. The actual type can be of any type that is a subclass of the declared type.

Type conversion, in conjunction with inheritance, polymorphism and dynamic binding, allows the behavior of an object to be different with different actual types. Type conversion sometimes happens automatically, and sometimes is forced by using an explicit cast operator. Three new class mutation operators are introduced for type conversion to insert, delete, and change type cast operators. In the definitions below, the class `Child` inherits from the class `Parent` and `GrandChild` inherits from `Child`.

- **PCI – Type cast operator insertion:** The PCI operator changes the actual type of an object reference to the parent or to the child of the original declared type. The mutant will have different behavior when the object to be cast has hiding variables or overriding methods.

Original Statements		**PCI Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
pRef.toString();	Δ	((**Child**) pRef).toString();

- **PCD – Type cast operator deletion:** The PCD operator deletes type casting operators; the inverse of PCI.

Original Statements		**PCD Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
((**Child**) pRef).toString();	Δ	pRef.toString();

- **PCC – Cast type change:** The PCC operator changes the type to which an object reference is being cast. The new type must be in the type hierarchy of the declared type (that is, a valid cast).

Original Statements		**PCC Mutant**
GrandChild gRef;		GrandChild gRef;
((**Parent**) gref).toString();	Δ	((**Child**) gref).toString();

3. EQUIVALENCY CONDITIONS

Detecting equivalent mutants by hand is time-consuming, usually prohibitively so. Budd and Angluin [4] showed the general problem to be undecidable, but heuristics have been developed to partially solve the problem. In 1979, Baldwin and Sayward [2] proposed using compiler optimization techniques to detect equivalent mutants. These ideas were developed into algorithms by Offutt and Craft [16] and implemented in the Mothra toolset, detecting an average of 10% of the equivalent mutants for 15 program units. Offutt [8, 21] 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 [18] developed this idea in an automated tool, which detected an average of 45% of the equivalent mutants for the program units used in the previous paper [16], and found over 70% of **unreachable statements** in a separate group of program units. Hierons, Harman and Danicic [11] suggested using program slicing to detect equivalent mutants and extensively analyzed the differences between the slicing and constraint solving approaches.

Previous work was based on statement-level mutation operators, and the two tools were for Fortran (with Mothra).

This paper introduces specific techniques for Java class mutation operators that are adapted from constraint solving approaches. Instead of running in a "post-processing mode," after mutants are generated, as the previous tools did, MuJava integrates the equivalent mutation analysis with mutant generation, as suggested by Hierons, Harman and Danicic [11]. MuJava implements specific, focused, heuristics that avoid equivalent mutants for specific mutation operators. This approach is based on *equivalency conditions* for mutation operators, which in turn is based on the conditions under which mutants are killed. The following eight paragraphs define equivalency conditions for sixteen mutation operators.

IHI and IHD: In OO programs, a child class can override a variable declared in an ancestor by declaring a new variable with the same name and type. The new variable is called a *hiding variable*. The IHI mutation operator inserts hiding variable declarations and the IHD operator deletes hiding variable declarations. However, if the variable being hidden (in the ancestor class) has `private` access level, it cannot be accessed directly from the child class, thus all mutants that insert or delete hiding variables are equivalent.

Note that private variables can be accessed only by methods of the class where it is declared, and not by methods of subclasses. Consider the following example.

```
class Student {
    public String name = "Kim";
    public String studentID = "123";
    public String getName() {
        return name;
    }
}

class GraduateStudent extends Student {
    public String name = "Lee";        // variable hiding
    public String getNameAndID1() {
        return name + ": " + studentID;
    }
    public String getNameAndID2() {
        return getName() + ": " + studentID;
    }
}
```

The following statements contain an instantiation of `GraduateStudent` followed by several syntactically legal ways to refer to the variable's *name*.

```
GraduateStudent s = new GraduateStudent();
System.out.println (s.name);                  → Lee
System.out.println (s.getName());             → Kim
System.out.println (s.getNameAndID1());       → Lee 123
System.out.println (s.getNameAndID2());       → Kim 123
```

The example shows that the method *getName()* of the object *s* does not access the hiding variable but the hidden variable of the variable *name*.

IOR: When a method `f()` in a child class overrides method `f()` in the parent class, the IOR mutation changes the name of the method in the parent, including all references to the method in the parent. This means that the child class no longer overrides the method. However, if a method in a parent class is overridden, it will not be accessed by the child class, thus changing its name in the parent class has no effect unless the child accesses the parent's method using the `super` keyword. Otherwise, the IOR operator generates equivalent mutants.

AMC: The AMC operator usually creates useless mutants. If the access is strengthened (for example, public to private), the mutated program usually does not compile – the mutant tries to use a variable that is out of scope. If the access is weakened, the mutated program is often equivalent – the mutant can still use the same variables.

The only situation in which the AMC operator can create a killable mutant is when there is both overriding and overloading. Specifically, the change to the access level of an overloaded method blocks access to it, and another, overloaded, method exists with parameters of compatible type. When the first method is blocked, the overloaded method is executed instead. Consider the following example.

Original Statements	**AMC Mutant**
`public void f (float i);`	`public void f (float i);`
`public void f (int i);`	Δ **`private`** `void f (int i);`

The access level for *f (int i)* is changed to `private`. If another class has the call *f (3)*, the method *f (float i)* is executed instead of *f (int i)*. As it happens, this situation is also covered by the OMD operator, which deletes overloaded methods. Thus, the AMC operator generates uncompilable, equivalent, or redundant mutants, and is not needed in MuJava.

IOD: The IOD operator deletes methods that override methodgs in a class's parent to ensure that the method invocation actually invokes the intended method. If the overriding method is exactly the same as the method it overrides, the IOD operator generates an equivalent mutant. This happens when an overriding method is a *delegation method*, and only calls the parent's method. An example is:

```
void MethodName ()
{
   super.MethodName();
}
```

Deleting this method has no effect, so the mutant would be equivalent and is not created.

ISI and **ISD:** If a subclass hides a variable or method of one of its ancestors, it can still reference the hidden member by using the `super` keyword. The subclass can also use `super` to invoke a parents version of a method that has been overridden. The ISI operator inserts the `super` keyword when possible and the ISD operator deletes occurrences of `super`. However, if a reference is to a non-overridden member, adding or deleting `super` has no effect, because the ancestor's version is used in either case. Thus, ISI and ISD mutants on non-overridden members are equivalent. Programmers sometimes use the `super` keyword unnecessarily to better document the program.

IPC: When new objects of a subclass are created, the parent's default constructor (no arguments) is automatically called, then the subclass's constructor is called. Programmers sometimes explicitly add calls to a parent's constructor at the beginning of the subclass's constructor, usually to call a different constructor. Deleting an explicit call to the parent's **default constructor** produces an equivalent mutant, because the default constructor has already been called implicitly.

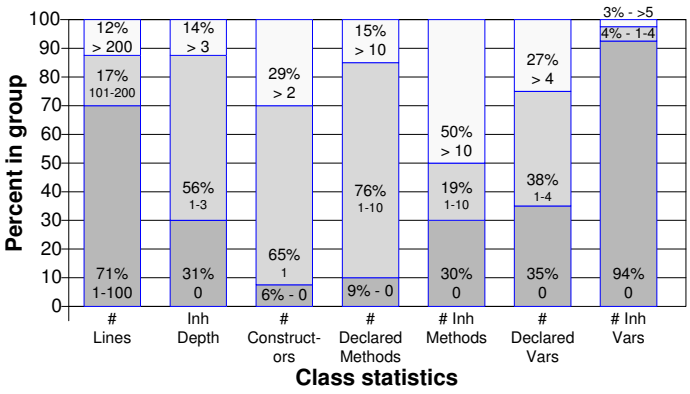

Figure 1: Summary of Statistics From the 866 Classes

JTI and **JTD:** The JTD operator deletes uses of the keyword `this`. Within a method body, uses of the keyword `this` refers to the current object. Methods can usually directly refer to the objects instance variables by name. However, if a variable is hidden by a local variable or parameter that has the same name, the object instance variable is hidden, but the method can access those variables through `this`.

Programmers occasionally use the keyword `this` unnecessarily to access instance variables that can be accessed directly. If the keyword `this` is inserted or deleted when the instance variable is not hidden, an equivalent mutant is generated.

PNC, **PMD**, **PPD**, **PCI**, **PCC**, and **PCD**: These mutation operators change a declared type or an actual type of a variable to a compatible type – its child class or super class. Just changing the type does not necessarily mean the behavior is changed. The only way that behavior can be changed is if the change in type results in different variables being used or different methods being called. This happens if some members are overridden, so that the original and mutated type refer to different members. If the same members are accessed in both original and mutated types, the mutant is equivalent.

3.1 Equivalent Detection Results

This subsection presents data from applying the equivalence detection tool to 866 classes from six open source applications. The applications varied in size from less than 100 to more than 300 classes. JavaCat is an application for managing files on different drives, JvFTP is a Java ftp client library, JMSN is a pure Java Microsoft MSN messenger clone, jEdit is a programming text editor, BCEL is a byte code engineering library for analyzing, creating, and manipulating bytecode, and JExcelApi is a Java library for reading, writing, and modifying Microsoft Excel spreadsheets. MuJava cannot, as yet, handle interfaces, abstract classes, and GUI classes thus were excluded.

Figure 1 summarizes descriptive statistics from the classes. Each stacked bar represents a different statistic, with data split into three groups depending on the statistic. The number of lines (first bar) is divided into classes with between 1 to 100, 101 to 200, and more than 200 lines. Most classes are relatively small; most have 100 lines or less and almost 90% have less than 200. The inheritance depth does not include `java.lang.Object`[1], so a class with no `extends` keywords has depth zero. Although space prohibits showing these data, the inheritance depths varied greatly by application. Most classes in JvFTP (over 90%) have inheritance depth of zero or one. BCEL and JExcelApi have more inheritance depth; BCEL does not reach the 90% level until inheritance depth 4, and JExcelApi does not reach the 90% level until inheritance depth 3.

Not surprisingly, most classes have only one or two constructors. Most classes have less than 10 methods and many classes have no methods at all. Most classes with no declared methods have constructors and otherwise use inherited methods, including used-defined exceptions. More than 90% of the classes use no inherited variables at all; primarily because their parents either have no variables or all their variables are declared to be `private`.

Operator	Mutants	Equiv	Percent
AMC	20,087	**20,080**	**(99.97%)**
IHI	492	**345**	**(70.12%)**
IHD	0	**0**	**(–)**
IOD	8,645	**19**	**(0.22%)**
IOR	638	**603**	**(94.51%)**
ISI	414	**392**	**(94.69%)**
ISD	46	**43**	**(93.48%)**
IPC	214	**19**	**(8.88%)**
PNC	882	**812**	**(92.06%)**
PMD	192	**155**	**(80.73%)**
PPD	412	**410**	**(99.51%)**
PCI	7,793	**4,267**	**(54.75%)**
PCC	710	**132**	**(18.59%)**
PCD	244	**220**	**(90.16%)**
JTI	14,991	**14,243**	**(95.01%)**
JTD	1,075	**660**	**(61.40%)**
Total	**56,835**	**42,400**	**(74.60%)**

Table 2: Number of Equivalent Mutants Detected

Table 2 shows the number of equivalent mutants found by MuJava for the 866 classes. These data are **dramatically different** from similar data for statement-level mutation operators [8, 17, 16, 19, 18, 9], which found between 5% and 15% equivalent mutants. On average, almost 75% of all class level mutants for sixteen mutation operators were identified to be equivalent, and over 90% for eight of them! Note that MuJava does not identify all equivalent mutants (the general problem is unsolvable) and equivalency conditions were not identified for the other 12 MuJava operators. That is, there are almost certainly more equivalent mutants than identified here.

4. COUNTING THE MUJAVA MUTANTS

The cost of testing with mutation depends on the number of mutants. For statement-level mutants, the number of mutants is on the order of the number of variable declarations times the number of variable references, typically a large number even for small software units. The *selective operator set* [17] reduced this to the order of the number of variable declarations; still hundreds of mutants for small program units. Mutation can generate hundreds of thousands of mutants for large programs.

[1]In Java, all classes implicitly inherit from the builtin `java.lang.Object`.

Operator	TOTAL
AMC	7
IHI	147
IHD	0
IOD	8,626
IOP	58
IOR	35
ISI	22
ISD	3
IPC	195
PNC	70
PMD	37
PPD	2
PCI	3,526
PCC	578
PCD	24
PRV	2305
OMR	270
OMD	7
OAC	13,443
JTI	748
JTD	415
JSI	2,249
JSD	1,058
JID	546
JDC	24
EOA	6
EOC	17
EAM	4,702
EMM	1,046
Total	40,159
Average (per class)	46.37

Table 3: Number of Class Mutants

This section examines how many mutants are generated with class mutation operators. First, we determine the average number of class mutants with the subject classes, then the number of mutants per class, and finally the number of mutants per mutation operator.

4.1 Mutant Generation for Classes

Table 3 shows the number of mutants that MuJava generated for each class operator for the six applications. These numbers are based on the mutants after equivalence detection was done. The total and the average numbers of mutants are listed at the bottom of the table. A total of $40,159$ mutants were generated for the six applications, and the mean number of mutants per class was 46.37. This is much smaller than can be expected with statement-level mutation operators, largely because the object-oriented language features that these operators modify are used less frequently than features such as arithmetic operators and variable references.

Some mutation operators produced far fewer mutants than others. AMC, ISD, and PPD produced less than ten mutants apiece, and OAC produced more than 10,000 mutants. There are two reasons for this. Either the language features the operators change are rarely used or most of their mutants are equivalent. These data indicate that OO language features are used with very different frequencies.

4.2 Mutant Generation Per Class

The number of statement-level mutants for the 866 classes is close to one million (as determined analytically based on Mothra's [8] operators), so Table 3 shows that class mutation operators create fewer mutants than statement-level operators do. Moreover, the number of mutants varied greatly among classes. Figure 2 summarizes the data by dividing the classes into five groups. Almost 11% of the classes did not produce any mutants. Most of these were very simple classes that did things such as implementing constructors without declaring any methods. It seems unlikely that a sophisticated testing technique such as mutation would be used on classes like this.

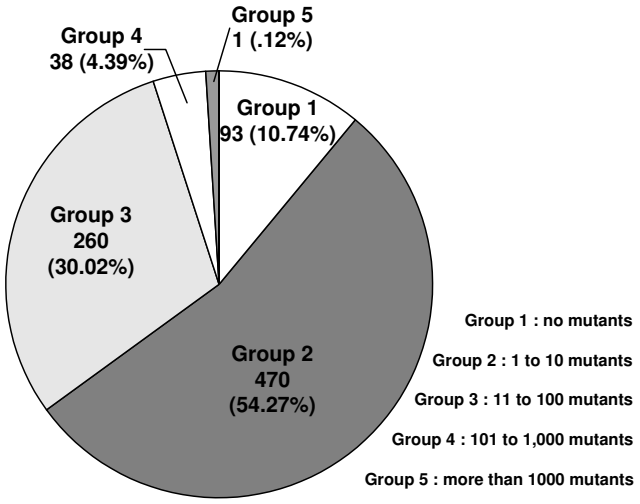

Figure 2: Number of Mutants Per Class, Grouped in Ranges

More than half of the classes produced less than ten mutants and slightly more than 30% of the classes produced between ten and one hundred mutants.

Only one class produced more than 1000 mutants (12,221 mutants to be exact). Among those 12,221 mutants, 12,180 were from one operator: OAC. This class calls a lot of overloaded methods defined in other classes, so is clearly a special case. So even though Table 3 shows that the most "prolific" mutation operator was OAC, with 13,443 total mutants for the 866 classes, 865 of those classes only produced a total of 1,263 OAC mutants. If we removed the one outlier class, OAC would no longer be the anomaly with more than 50% more mutants than the second place operator (IOD), it would only be the sixth most prolific operator. This demonstrates the wide variation in the number of mutants per class, and emphasizes the need to consider the number of mutants per operator.

4.3 Mutant Generation Per Operator

Figure 3 shows the number of mutants generated per class, summarized into groups as in Figure 2. The stacked bars summarize a lot of data. The first column, AMC, indicates that 99.19% of the classes were in group 1, that is, had no AMC mutants. Only .81% of the classes had betwen 1 and 10 AMC mutants, and no classes had more than 10. Note the scaling in Figure 3; there is a jump from 0 to 40 and the numbers from 90 to 100 are shown with more precision. This is done to conserve space and to highlight thenumber of mutants for classes in groups 2 through 5.

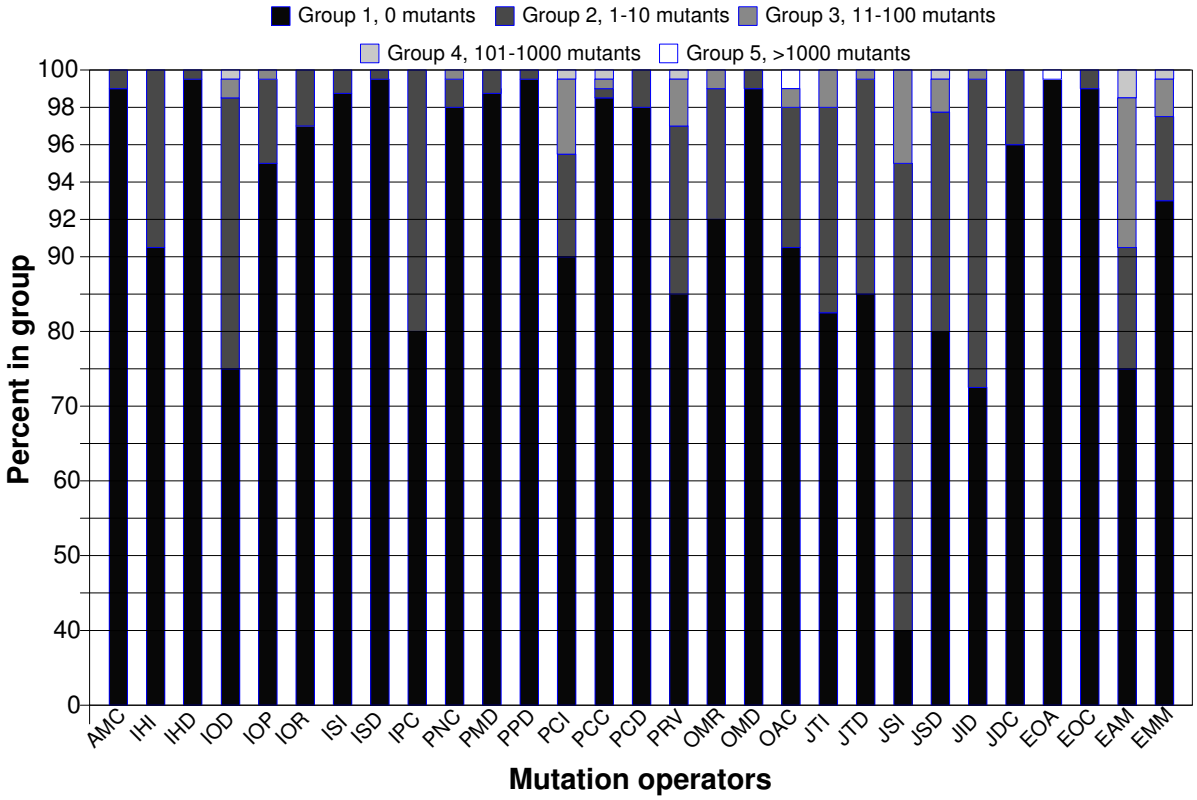

Figure 3: Number of Mutants per Operator per Class, Grouped

The information that stands out from these data is that 20 of the 29 mutation operators generated no mutants for over 90% of the classes. Eight operators (AMC, IHD, ISD, PPD, PCC, OMD, EOA and EOC) generated no mutants for over 99% of the classes. Only one operator, JSI, generated at least one mutant for more than half of the classes. Figure 4 illustrates this information by charting the number of mutation operators that produced zero mutants for 90% or more of the classes, 80% or more, etc. It is premature to conclude that mutation operators that produce few mutants are not useful; these data probably reflect the fact that some OO language features simply are not used very often. When they are used, something like mutation should be used to test their use. We suggest that these data show that the execution cost of class mutation testing is manageable and practical.

An immediate determination from the data in Table 3 and Figure 3 is that the OAC operator presents significant but rare performance risks. It produced no mutants at all in over 90% of the class, and only one mutant in 8%, but over 1,000 mutants for one class! Thus, an argument could be made to remove OAC.

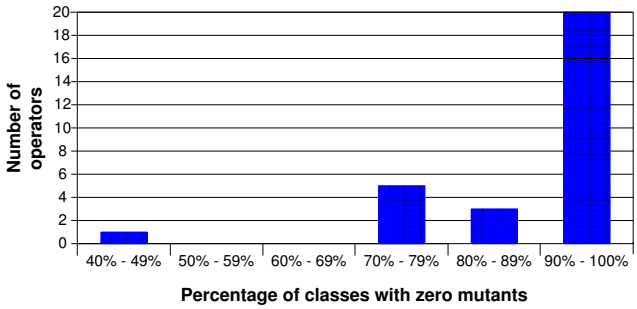

Figure 4: Number of Mutation Operators that Produced Zero Mutants

5. CONCLUSIONS AND FUTURE WORK

This paper presents a method for determining equivalent class-level mutants, data on the number of equivalent mutants found, and data on the number of mutants created. The automated Java mutation system MuJava was used to investigate the characteristics of class-level mutants generated from 866 classes drawn from six open-source Java programs.

The *equivalency conditions* described in this paper found that more than 70% of the class-level mutants were equivalent, far more than the 5% to 15% found with unit-level mutants. Results on the open source software show that there are many fewer class-level mutants than unit-level mutants (an average of 46 per class). We conclude that mutation testing, which is often considered to be too expensive for practical use in unit testing, is much more practical at the inter-class testing level.

In the future, we plan to determine selective class-level mutation operators, as was done with statement-level operators [17]. This will require a systematic evaluation of each operator against a number of classes, and preliminary work on this problem has already started. We also plan to carry

out fault detection experiments, both in laboratory settings and with industrial case studies.

An eventual goal is to add one more level of automation to class-level mutation testing, specifically the ability to automatically generate tests to kill most of the mutants. There are several differences from unit-level automatic test data generation. The most obvious is that test cases are more complicated for inter-class testing – sequences of method calls on instantiated objects. A related difference is that the controllability problem [3] is more severe with inter-class testing. Finding values for parameters to test public methods is hard enough, but private methods can only be called indirectly through public methods, and reaching specific statements inside private methods is beyond the abilities of the strongest automatic test data generation research tools.

6. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In A. Press, editor, *Proc. of the 27th International Conference on Software Engineering*, pages 402–411, St. Louis, MO, USA, May 2005. ACM Press. SESSION: Empirical evaluation of testing.

[2] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical Report 161, Yale University, Dept. of Computer Science, 1979.

[3] R. V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, 6(3/4):125–252, 1996.

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

[5] 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)*, pages 267–270, Macau SAR, China, December 2001. IEEE Computer Society.

[6] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, pages 1–14, December 2002.

[7] 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.

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

[9] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, September 1997.

[10] L. Gallagher and J. Offutt. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, 2006. Accepted for publication.

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

[12] S. Kim, J. Clark, and J. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Net.ObjectDays Conference on Object-Oriented Software Systems*, October 2000.

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

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

[15] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava home page. online, 2005. http://ise.gmu.edu/∼offutt/mujava/, http://salmosa.kaist.ac.kr/LAB/MuJava/, last access November 2005.

[16] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification, and Reliability*, 4(3):131–54, September 1994.

[17] A. 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.

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

[19] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software-Practice and Experience*, 26(2):165–176, February 1996.

[20] 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.

[21] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software–Practice and Experience*, 29(2):167–193, January 1999.