Statistical Investigation on Class Mutation Operators

Yu-Seung Ma, Yong-Rae Kwon, and Sang-Woon Kim

Although mutation testing is potentially powerful, it is a computationally expensive testing method. To investigate how we can reduce the cost of object-oriented mutation testing, we have conducted empirical studies on class mutation operators. We applied class mutation operators to 866 classes contained in six open-source programs. An analysis of the number and the distribution of class mutants generated and preliminary data on the effectiveness of some operators are provided. Our study shows that the overall number of class mutants is smaller than for traditional mutants, which offers the possibility that class mutation can be made practically affordable.

Keywords: Mutation testing, object-oriented, class mutation operator.

Manuscript received June 23, 2008; revised Jan. 21, 2009; accepted Feb. 17, 2009.

This work was supported by the IT R&D program of MKE/IITA, Rep. of Korea (2008-S-023-01, Development of NanoQplus-Based Sensor Network Simulator).

Yu-Seung Ma (phone: +82 42 860 6551, email: ysma@etri.re.kr) is with Software & Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

Yong Rae Kwon (email: kwon@cs.kaist.ac.kr) and Sang-Woon Kim (email: swkim@salmosa.kaist.ac.kr) are with the Computer Science Department, KAIST, Daejeon, Rep. of Korea.

I. Introduction

Mutation testing [1], [2] is a fault-based testing technique that measures the effectiveness of test cases. Faults are introduced into the program by creating a set of faulty versions called mutants. Mutation operators describe syntactic changes to a program and are applied to the original program to create mutants. A mutant is “killed” by a test case that causes the mutant program to produce an output different from that produced by the original program. An *equivalent mutant* is a mutant program that is functionally equivalent to the original program and therefore cannot be killed by any test case. A test case that kills a nonequivalent mutant is considered to be effective at finding faults in the program. The *mutation score*, which represents the quality of a test set, is calculated as the ratio of the number of killed mutants to the number of nonequivalent mutants.

Although mutation testing is powerful, the cost of applying mutation testing is extremely high [3], [4] because quite a large number of mutants are produced and executed. As a result, a lot of research [3], [5]-[7] has been conducted to reduce the number of generated mutants. However, most research has focused on traditional mutation operators.

Class mutation operators are substantially different from traditional mutation operators. This paper presents results from two empirical studies designed to investigate the properties of class mutation operators. For the studies, we applied class mutation operators to several open source software packages and gathered data on their applicability and usefulness.

The first empirical study is designed to examine how many mutants are generated with class mutation operators. Class mutation operators modify object-oriented features, such as inheritance, polymorphism, dynamic binding, and encapsulation. These are high-level language features that are used less frequently than arithmetic operators and variable references; thus, fewer mutants are expected to be generated

© 2009 ETRI

with class mutation operators than with traditional mutants. This study quantifies this observation and finds a detailed distribution of the number of class-level mutants.

The second study is designed to investigate whether class mutation operators that generate very few mutants are necessary. The study begins by questioning whether we should spend time and labor to produce so few mutants. If those mutants model faults that are detected easily, the mutation operators that generate so few mutants can be ignored in mutation testing.

The remainder of this paper is organized as follows. Section II briefly describes class mutation operators. Section III summarizes our subject applications and gives details regarding the number and distribution of class mutants. Section IV, based on the results from the previous section, discusses the necessity of class mutation operators that produce few mutants. Conclusions and future work are presented in section VI.

II. Background

Mutation operators have been designed for various programming languages, including Fortran IV, COBOL, Fortran 77, C, Lisp, Ada, and Java. Because mutation operators are defined in terms of the grammar of a particular language, they must be separately designed for each language. These program-based mutation operators can be classified into the following three groups.

- **Traditional mutation operators** [8], [9]**:** Mutation operators were originally developed for procedural languages. They are called traditional mutation operators to distinguish them from other kinds of mutation operators. Traditional mutation operators handle basic elements of a programming language such as operands and operators.
- **Interface mutation operators** [10]: They were proposed to evaluate how well the interactions between various units have been tested. They are restricted in their application to only those parts of a code that are related to module interactions, such as function calls, parameters, and global variables.
- **Class mutation operators** [11], [12]: Class mutation operators were developed because traditional mutation operators are not sufficient to test object-oriented programs. They handle object-oriented specific features such as inheritance, polymorphism, and dynamic binding.

This paper targets mutation operators used in MuJava [12], [13], a widely used mutation tool for Java. MuJava automatically generates mutants, runs the mutants against tests supplied by the tester, and reports the mutation score of the test set. Tables 1 and 2 show traditional and class mutation operators used in MuJava.

The traditional mutation operators listed Table 1 modify expressions by replacing, deleting, or inserting primitive operators of Java. The class mutation operators in Table 1 are grouped into four categories: encapsulation, inheritance, polymorphism, and Java-specific features. The first three groups are related to language features that are common to all object-oriented programming languages. The last group is related to object-oriented features specific to Java.

Detailed definitions for the operators may be found in previous papers [12], [14] and on the MuJava website [13].

Table 1. Traditional mutation operators for Java.

Operator	Description
AOR	Arithmetic operator replacement
AOI	Arithmetic operator insertion
AOD	Arithmetic operator deletion
ROR	Relational operator replacement
COR	Conditional operator replacement
COI	Conditional operator insertion
COD	Conditional operator deletion
SOR	Shift operator replacement
LOR	Logical operator replacement
LOI	Logical operator insertion
LOD	Logical operator deletion
ASR	Assignment operator replacement

III. Analysis of Class Mutation Generation

The cost of mutation testing depends on the number of mutants. For traditional 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. For example, Mothra [8], [15], the Fortran mutation system, generated 951 traditional mutants for the commonly studied 30-line triangle classification program TriTyp [16]. The selective operator set [7] reduced this to the order of the number of variable declarations. Still, hundreds of mutants are produced for small program units.

In this section, we examine the number of class mutants generated when the mutation operators are applied to the subject classes with MuJava [12], [13]. First, we briefly describe the subjects used in the experiments. Then, we present the analysis results obtained from the experimental data.

1. Applications Used

Due to the extremely high cost of mutation testing, most research results on mutation testing were obtained by applying

Table 2. Class mutation operators for Java.

Language feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHI	Hiding variable insertion
	IHD	Hiding variable deletion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-specific features	JTI	This keyword insertion
	JTD	This keyword deletion
	JSI	Static modifier insertion
	JSD	Static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

mutation to between 10 to 20 methods (in the case of procedural programs) or classes (in the case of object-oriented programs). To ensure that the analysis results are as valid as possible, we decided that at least several hundred classes should be employed in the experiment. In our experiments, we used applications of various sizes, ranging from applications containing less than 100 classes to applications containing more than 300 classes. We chose open source programs in

Table 3. Six subject applications.

Application	Description
JavaCat[a)]	Application for managing files on different drives (CD-ROMs, hard disks, and so on)
JvFTP[b)]	Java ftp client library
JMSN[c)]	Pure Java Microsoft MSN messenger clone
JEdit[d)]	Programmer's text editor
BCEL[e)]	Byte code engineering library for analyzing, creating, and manipulating byte code
JexcelApi[f)]	Java library for reading, writing, and modifying Microsoft Excel spreadsheets

a) http://javacat.sourceforge.net b) http://jvftp.sourceforge.net
c) http://jmsn.sourceforge.net d) http://www.jedit.org
d) http://jakarta.aphache.org/bcel f) http://jexcelapi.sourceforge.net

diverse domains. Table 3 summarizes the six open source applications chosen for our studies. The URLs to their websites are supplied in footnotes. The six open-source applications have a total of 866 classes with interfaces, excluding abstract classes and GUI classes. Our experiment excluded those classes because MuJava does not support them.

Table 4 lists descriptive statistics on the 866 classes, summarizing the data into groups. For example, the number of lines is divided into six groups: between 1 to 100, 101 to 200, 201 to 300, 301 to 400, 401 to 500, and more than 500 lines. In counting the inheritance depth, java.lang.Object[1)] was not included, so a class without the "extend" keyword has a depth of zero. The inheritance depths varied from application to application. However, most classes have an inheritance depth less than three, only one or two constructors, and less than 10 methods. It is interesting to note that there are many classes that have no declared methods at all. Classes with no declared methods merely implement constructors and use methods inherited from their parent classes. While inherited methods are used quite extensively, we found that many of the classes do not inherit variables at all primarily because their parents have no variables or declare all their variables to be private.

2. Analysis Result

This subsection analyzes class mutants generated with the 866 classes. First, we present the number of equivalent mutants detected by MuJava. Then, we examine how many class mutants are generated with those equivalent mutants excluded.

A. Equivalent Mutants Detected by MuJava

MuJava has an ability to detect equivalent mutants for sixteen class mutation operators listed in Table 5. Table 5

1) The root of the Java class hierarchy. Every class in the Java system has java.lang.Object as its ultimate parent.

Table 4. Summary of statistics from the 866 classes.

Application information		Number of classes						
		JavaCat	JvFTP	JMSN	jEdit	BCEL	JExcelApi	Total
Number of lines	1 - 100	7 (77.78%)	4 (28.57%)	23 (52.27%)	109 (55.61%)	213 (80.68%)	257 (75.81%)	613 (70.78%)
	101 - 200	2 (22.22%)	6 (42.86%)	15 (34.09%)	47 (23.98%)	33 (12.50%)	47 (13.86%)	150 (17.32%)
	201 - 300	0 (0.00%)	2 (14.29%)	3 (6.82%)	16 (8.17%)	8 (3.03%)	15 (4.42%)	44 (5.08%)
	301 - 400	0 (0.00%)	0 (0.00%)	2 (4.55%)	3 (1.53%)	3 (1.14%)	5 (1.48%)	13 (1.50%)
	401 - 500	0 (0.00%)	1 (7.14%)	0 (0.00%)	7 (3.57%)	1 (0.38%)	3 (0.89%)	12 (1.39%)
	501-	0 (0.00%)	1 (7.14%)	1 (2.27%)	14 (7.14%)	6 (2.27%)	12 (3.54%)	34 (3.93%)
Inheritance depth	0	2 (22.23%)	9 (64.29%)	33 (75.00%)	48 (44.39%)	48 (18.18%)	88 (25.96%)	267 (30.83%)
	1	3 (33.33%)	4 (28.57%)	7 (15.90%)	92 (46.94%)	61 (23.11%)	54 (15.93%)	221 (25.52%)
	2	1 (11.11%)	1 (7.14%)	2 (4.55%)	12 (6.12%)	101 (38.26%)	140 (41.30%)	257 (29.68%)
	3	0 (0.00%)	0 (0.00%)	2 (4.55%)	5 (2.55%)	38 (14.39%)	38 (11.21%)	83 (9.58%)
	4-	3 (33.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	16 (6.06%)	19 (5.60%)	38 (4.39%)
Number of constructors	0	0 (0.00%)	0 (0.00%)	12 (27.27%)	26 (13.27%)	8 (3.03%)	7 (2.06%)	53 (6.12%)
	1	9 (100.0%)	10 (71.42%)	23 (52.28%)	142 (72.45%)	142 (53.79%)	234 (69.03%)	560 (64.67%)
	2	0 (0.00%)	0 (0.00%)	8 (18.18%)	13 (6.63%)	76 (28.79%)	71 (20.94%)	168 (19.40%)
	3	0 (0.00%)	2 (14.29%)	1 (2.27%)	10 (5.10%)	34 (12.88%)	13 (3.84%)	60 (6.93%)
	4	0 (0.00%)	2 (14.29%)	0 (0.00%)	1 (0.51%)	2 (0.76%)	10 (2.95%)	15 (1.73%)
	5-	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (2.04%)	2 (0.76%)	4 (1.18%)	10 (1.15%)
Number of declared methods	0	1 (11.11%)	2 (14.29%)	3 (6.82%)	13 (6.63%)	11 (4.17%)	46 (13.57%)	76 (8.78%)
	1 - 10	7 (77.78%)	7 (50.00%)	28 (63.64%)	138 (70.41%)	224 (84.85%)	252 (74.34%)	656 (75.75%)
	11 - 20	1 (11.11%)	3 (21.42%)	9 (20.45%)	29 (14.80%)	15 (5.68%)	27 (7.97%)	84 (9.70%)
	21 - 30	0 (0.00%)	0 (0.00%)	3 (6.82%)	6 (3.06%)	3 (1.14%)	5 (1.47%)	17 (1.96%)
	31 - 40	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (2.04%)	6 (2.27%)	5 (1.47%)	15 (1.73%)
	41-	0 (0.00%)	2 (14.29%)	1 (2.27%)	6 (3.06%)	5 (1.89%)	4 (1.18%)	18 (2.08%)
Number of inherited methods	0	2 (22.22%)	9 (64.29%)	33 (75.00%)	84 (42.86%)	47 (17.80%)	88 (25.96%)	263 (30.37%)
	1	0 (0.00%)	0 (0.00%)	0 (0.00%)	18 (9.18%)	1 (0.38%)	147 (43.36%)	166 (19.17%)
	2	1 (11.11%)	4 (28.57%)	4 (9.09%)	40 (20.41%)	31 (11.74%)	74 (21.83%)	154 (17.78%)
	3	2 (22.22%)	0 (0.00%)	1 (2.27%)	44 (22.45%)	161 (60.99%)	25 (7.38%)	233 (26.90%)
	4	0 (0.00%)	0 (0.00%)	1 (2.27%)	5 (2.55%)	13 (4.92%)	4 (1.18%)	23 (2.66%)
	5-	4 (44.45%)	1 (7.14%)	5 (11.37%)	5 (2.55%)	11 (4.17%)	1 (0.29%)	27 (3.12%)
Number of declared variables	0	2 (22.22%)	2 (14.29%)	9 (20.46%)	40 (20.41%)	162 (61.36%)	90 (26.55%)	305 (35.22%)
	1	1 (11.11%)	0 (0.00%)	5 (11.36%)	28 (14.29%)	39 (14.77%)	40 (11.80%)	113 (13.05%)
	2	1 (11.11%)	2 (14.29%)	4 (9.09%)	26 (13.27%)	27 (10.23%)	54 (15.93%)	114 (13.16%)
	3	2 (22.22%)	4 (28.57%)	7 (15.91%)	14 (7.14%)	8 (3.03%)	29 (8.55%)	64 (7.39%)
	4	0 (0.00%)	1 (7.14%)	7 (15.91%)	14 (7.14%)	10 (3.79%)	14 (4.13%)	46 (5.31%)
	5-	3 (33.34%)	5 (35.71%)	12 (27.27%)	74 (37.76%)	18 (6.82%)	112 (33.04%)	224 (25.87%)
Number of inherited variables	0	4 (44.45%)	12 (85.72%)	37 (84.09%)	186 (94.90%)	256 (96.97%)	320 (94.40%)	815 (94.11%)
	1	1 (11.11%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (1.18%)	5 (0.58%)
	2	0 (0.00%)	0 (0.00%)	1 (2.27%)	1 (0.51%)	0 (0.00%)	4 (1.18%)	6 (0.69%)
	3	1 (11.11%)	0 (0.00%)	2 (4.55%)	3 (1.53%)	0 (0.00%)	4 (1.18%)	10 (1.15%)
	4	0 (0.00%)	1 (7.14%)	3 (6.82%)	0 (0.00%)	0 (0.00%)	1 (0.29%)	5 (0.58%)
	5-	3 (33.33%)	1 (7.14%)	1 (2.27%)	6 (3.06%)	8 (3.03%)	6 (1.77%)	25 (2.89%)
Total classes		9	14	44	196	264	339	866

Table 5. Number of equivalent mutants detected by MuJava.

Operator	Number of class mutants		
	Total	Equivalent	Remainder
AMC	20,087	20,080 (99.97%)	7 (0.03%)
IHI	492	345 (70.12%)	147 (29.88%)
IHD	0	0 (-)	0 (-)
IOD	881	42 (4.77%)	839 (95.23%)
IOR	638	603 (94.51%)	35 (5.49%)
ISI	414	392 (94.69%)	22 (5.31%)
ISD	46	43 (93.48%)	3 (6.52%)
IPC	214	19 (8.88%)	195 (91.12%)
PNC	882	812 (92.06%)	70 (7.94%)
PMD	192	155 (80.73%)	37 (19.27%)
PPD	412	410 (99.51%)	2 (0.49%)
PCI	7,793	4,267 (54.75%)	3,526 (45.25%)
PCC	710	132 (18.59%)	578 (81.41%)
PCD	244	220 (90.16%)	24 (9.84%)
JTI	14,991	14,243 (95.01%)	748 (4.99%)
JTD	1,075	660 (61.40%)	415 (38.60%)
Total	49,071	42,423 (86.45%)	6,648 (13.55%)

shows the number of equivalent mutants detected by MuJava for the 866 classes. The data of Table 5 is dramatically different from similar data for traditional mutation operators, which found between 5% and 15% equivalent mutants. On average, almost 86% of all class mutants for the 16 mutation operators were identified to be equivalent, and over 90% for eight of them. Note that MuJava does not identify all equivalent mutants and does not handle equivalent mutants for the other 12 operators. That is, there are almost certainly more equivalent mutants than are identified here.

More detailed information about the equivalent detection method of MuJava can be found in [14]. Note that the result of IOD mutants differs from the result in [14]. We found that MuJava used in [14] has some errors in handling IOD mutants. We recalculated IOD mutants after fixing the errors.

B. Mutant Generation per Application

Table 6 shows the number of class mutants generated for each of the six applications. The total and the average numbers of mutants are listed at the bottom row of the tables. A total of 32,379 class mutants were generated for the six applications, and the mean number of class mutants per class was 37.39.

For comparison with traditional mutation operators, we also generated traditional mutants for the same target application, whose results are shown in Table 7. The number of traditional mutants was about twice the number of class mutants. It was somewhat surprising to find that the number of traditional mutants was noticeably less than with the case of procedural programs. Considering that the average line number of the total 866 classes was 124.6, the number of traditional mutants turned out to be much smaller than expected. From this data, we conclude that the cost of mutation testing for object-oriented programs may be much less than the cost for procedural programs.

A detailed analysis of the numbers of mutants generated is more illuminating. Let us call program elements that mutation operators modify, including operators, expressions, and statements *mutation targets* and let us call the mutated elements that replace the targets *mutation variants*. The number of mutants is the product of the number of mutation targets and the number of mutation variants. Note that this formula is valid only for first-order mutants, mutants into which a single fault is injected. For example, the number of mutation targets for the ISD operator is the number of occurrences of the keyword "super," and it only has one mutation variant, deleting the super keyword. Therefore, the number of ISD mutants is the number of occurrences of the "super" keyword. As another example, the JDC operator deletes programmer supplied default constructors, so the JDC operator has one mutation target and one mutation variant. On the other hand, the OAC operator changes the order and number of arguments in method calls when there is an overloading method that matches the new argument list. Thus, all combinations of the parameters are possible mutation variants.

From this analysis, we conclude that there must be fewer class mutants than traditional mutants because the numbers of mutation targets and mutation variants of class mutation operators are usually small. Some class mutation operators such as ISD and PPD produced few mutants. However, it may be premature to conclude that mutation operators that produce few mutants are not useful. This data probably reflects the fact that some object-oriented language features simply are not utilized that often.

C. Mutant Generation per Class

This subsection further explores the number of class mutants generated per class. Figure 1 summarizes the data by dividing the classes into five groups. Although the mean number of class mutants per class was 37.39, more than half of the classes (group 2) produced less than ten mutants. Slightly more than 30% of the classes (group 3) produced between ten and one hundred mutants.

Almost 11% of the classes (group 1) did not produce any mutants. Most of the classes in group 1 were very simple classes that took actions such as implementing constructors without declaring any methods. It seems unlikely that a

Table 6. Total number of class mutants with six Java applications.

Operator	Number of class mutants						
	JavaCat (9 classes)	Jvftp (14 classes)	Jmsn (44 classes)	jEdit (196 classes)	BCEL (264 classes)	JExcelApi (339 classes)	Total (866 classes)
AMC	0	0	0	1	6	0	7
IHI	1	2	8	16	105	15	147
IHD	0	0	0	0	0	0	0
IOD	6	18	23	190	546	56	839
IOP	2	0	1	5	47	3	58
IOR	3	0	3	12	14	3	35
ISI	3	7	1	9	2	0	22
ISD	0	0	0	2	0	1	3
IPC	2	0	1	16	167	9	195
PNC	0	0	0	59	3	8	70
PMD	0	0	0	1	0	36	37
PPD	0	0	0	0	0	2	2
PCI	0	0	0	646	2,445	435	3,526
PCC	0	0	0	0	571	7	578
PCD	0	0	0	13	8	3	24
PRV	5	51	119	1,641	99	390	2,305
OMR	0	10	32	118	27	83	270
OMD	0	0	0	1	6	0	7
OAC	0	11	51	218	61	13,102	13,443
JTI	6	37	59	262	351	33	748
JTD	4	31	32	117	212	19	415
JSI	29	33	107	731	273	1,076	2,249
JSD	0	17	30	215	13	783	1,058
JID	7	35	105	113	92	194	546
JDC	1	0	3	11	2	7	24
EOA	0	0	0	2	4	0	6
EOC	0	0	0	10	5	2	17
EAM	105	221	262	1,329	979	1,806	4,702
EMM	244	10	61	250	127	354	1,046
Total	418	483	898	5,988	6,165	18,427	32,379
Average (per class)	46.44	34.50	20.41	30.55	23.35	54.36	37.39

Table 7. Total number of traditional mutants with six Java applications.

Operator	Number of traditional mutants						
	JavaCat (9 classes)	Jvftp (14 classes)	Jmsn (44 classes)	jEdit (196 classes)	BCEL (264 classes)	JExcelApi (339 classes)	Total (866 classes)
AOR	52	150	275	1	1,123	4,604	6,205
AOI	81	622	1,263	218	8,765	16,526	27,475
AOD	1	12	9	262	42	40	366
ROR	30	387	594	117	2,211	3,975	7,314
COR	0	58	80	731	490	596	1,955
COD	2	22	28	215	222	241	730
COI	17	199	283	113	1,361	2,283	4,256
SOR	0	0	16	11	0	98	125
LOR	0	4	20	2	66	366	458
LOI	27	219	385	10	3,098	4,824	8,563
LOD	0	1	0	1,329	4	1	1,335
ASR	0	8	40	250	84	970	1,352
Total	210	1,682	2,993	3,259	17,466	34,524	60,134
Average (per class)	23.33	120.14	68.02	16.63	66.16	101.84	69.44

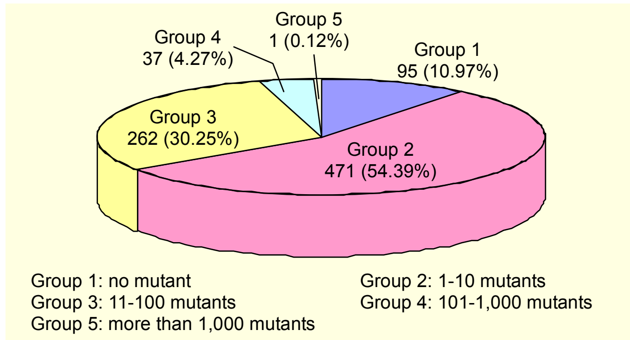

Fig. 1. Number of class mutants per class.

Table 8. Number of class mutants per source line.

Number of lines	Number of classes	Average number of class mutants
1 - 100	613	5.59
101 - 200	150	27.02
201 - 300	44	96.57
301 - 400	13	76.00
401 - 500	12	49.50
501 - 1,000	27	616.81
1,001 - 2,000	4	343.25
2,001 - 3,000	2	111.50
3,001 - 4,000	1	819.00
Total	866	37.39

sophisticated testing technique such as mutation would be used on such classes. Only one class produced more than 1,000 mutants (12,221 mutants to be exact). Surprisingly, among those 12,221 mutants, 12,180 were from one operator, namely, OAC. This class calls a lot of overloaded methods defined in other classes, so is clearly a special case. Therefore, even though Table 4 shows that the most prolific mutation operator was OAC with 13,443 total mutants for the 866 classes, 865 of those classes produced only a total of 1,263 OAC mutants. If we removed the one outlier class, OAC would only be the fifth most prolific operator. This demonstrates the wide variation in the number of class mutants per class, and guides us to consider the number of mutants per operator.

Table 8 shows the distribution of the number of class mutants according to the number of source lines. We classified the number of source lines into 9 groups and listed the average number of classes and the average number of class mutants per group.

Please note that programs whose sizes are between 501 and 1,000 lines produced high average numbers because classes which produced more than 10,000 mutants were included in

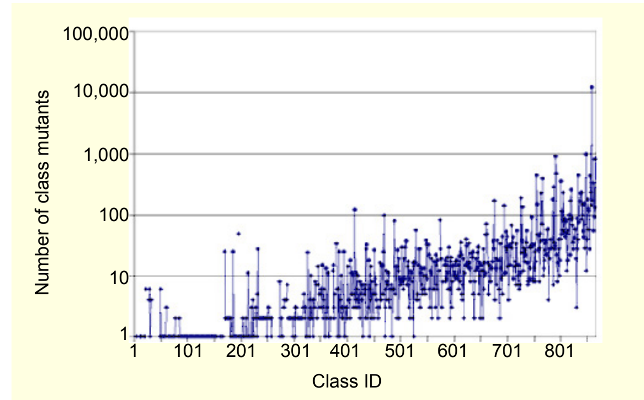

Fig. 2. Number of class mutants against class size.

the range. The exact number of lines of the class was 965. Excluding this particular class, the average number of the range becomes 170.50.

Figure 2 shows the number of class mutants for the 866 classes against the class size. That is, class ID 1 means the smallest class and class ID 866 means the biggest class. The result shows that the number of class mutants tends to increase as the number of code lines increases.

IV. Initial Attempt to Eliminate Unnecessary Mutation Operators

The final set of data to be presented was obtained in an effort to eliminate mutation operators that are not useful. As a first step, we analyzed mutation operators that generate very few mutants and raise the question: Should we spend time and labor to produce so few mutants?

As a starting point, we chose the six mutation operators that produced less than ten mutants in the 866 classes studied in this paper as shown in Table 6. These operators are AMC, IHD, ISD, PPD, OMD, and EOA.

1. Experimental Procedure

For each mutation operator, OP, from the set {AMC, IHD, ISD, PPD, OMD, EOA}, carry out the following steps.

Step 1. Choose the classes for which *OP* generated at least one mutant. This set is labeled C_{OP}.

Step 2. Eliminate equivalent mutants by hand. This is necessary because MuJava cannot eliminate equivalent mutants completely, and equivalent mutants would introduce erroneous data.

Step 3. For each class in C_{OP}, generate mutants with all operators except *OP*, that is, for the operators *M-OP*, where *M* is the set of all mutation operators. Generate tests to invoke methods of a class under test at least once and to kill all

mutants from *M-OP*. These tests are called $\overline{OP}$ - *adequate*. To eliminate any bias that could be introduced by one particular test set, we generated ten $\overline{OP}$ - *adequate* test sets for each class. In this experiment, a test set means a set of test cases whose form is a sequence of method calls. All of our results are average scores over the ten test sets.

Step 4. Run each $\overline{OP}$ - *adequate* test set against the mutants generated by *OP* and count how many *OP* mutants were killed. If most *OP* mutants were killed (a high mutation score), then *OP* can be considered to be redundant; therefore, eliminate it. Otherwise, *OP* produces mutants that need to be considered during testing, and *OP* should not be eliminated. In this experiment, we decided to use a threshold of 90% mutation score.

2. Experimental Target

Among the 866 classes, 11 Java classes turned out to generate at least one mutant for the six mutation operators: AMC, IHD, ISD, PPD, OMD, and EOA. Table 9 shows the number of class mutants for those 11 classes. For notational convenience, classes are renamed as c1, c2, c3, and so on. The "other operators" column gives the number of mutants that are generated with class mutation operators other than AMC, IHD, ISD, PPD, OMD, and EOA.

Although the classes shown in Table 9 have hundreds of source lines, they generated mutants for only one operator among the IHD, ISD, PPD, and EOA operators.

Table 9. Target class mutation operators.

Class	No. of lines	No. of methods	Number of class mutants							
			AMC	IHD	ISD	PPD	OMD	EOA	Others	Total
c1	114	11	0	0	1	0	0	0	1	2
c2	190	7	0	0	1	0	0	0	24	25
c3	95	8	0	0	1	0	0	0	14	15
c4	312	10	0	0	0	1	0	0	76	77
c5	760	30	0	0	0	1	0	0	55	56
c6	578	39	1	0	0	0	1	0	147	149
c7	242	18	5	0	0	0	5	0	25	35
c8	552	21	1	0	0	0	1	0	229	231
c9	164	5	0	0	0	0	0	2	130	132
c10	229	13	0	0	0	0	0	1	25	26
c11	552	34	0	0	0	0	0	3	72	75

Table 10. Mutation score of $\overline{AMC}$ - *adequate* test sets.

Class	No. of non-equivalent AMC mutants	Average number of test cases	Average mutation score
c6	1	57	100%
c7	5	20	100%
c8	1	48	100%
Average	2.33	41.67	100%

3. Experimental Result

This subsection describes results obtained by applying the previously mentioned procedure to each of the six mutation operators that produced the fewest mutants. The goal is to decide if each operator can be eliminated.

A. AMC

In [14], we found that the AMC operator usually creates uncompilable mutants, equivalent mutants, or mutants simulated by the OMD operator. With MuJava, which does not generate AMC mutants if they are expected to be uncompilable or equivalent, 7 AMC mutants are generated with the 866 target classes. It was noted that the AMC mutants are generated whenever OMD mutants are generated. Also, the AMC mutants are generated in the same numbers as the OMD mutants.

Table 10 gives the average mutation score of ten $\overline{AMC}$ - *adequate* test sets over the AMC mutants. The $\overline{AMC}$ - *adequate* test sets kill all of the AMC mutants. On investigating the source of the AMC mutants, we found that the AMC operator is simulated by the OMD operator. Therefore, we conclude that the AMC operator should be eliminated.

B. IHD

No IHD mutants were generated for any of the 866 classes studied. Although the IHD mutation operator is considered useless with our target systems, we need to find if there is a system that produces mutants for the IHD operator.

C. ISD

Table 11 gives the average mutation score of $\overline{ISD}$ - *adequate* test sets over the ISD mutants. The ISD mutant of the c2 class in Table 9 turned out to be equivalent; therefore, the c2 class is excluded from Table 11.

For the c1 class, the average mutation score is 50%. Five out of ten $\overline{ISD}$ - *adequate* test sets kill the ISD mutant. The $\overline{ISD}$ - *adequate* test sets had an average mutation score of 80% for the c3 class.

We examined the conditions under which the ISD mutant of the first class can be killed. Recall that the ISD operator deletes occurrences of the keyword "super." The c1 class has three constructors, and the ISD mutant was killed if and only if one

Table 11. N Mutation score of $\overline{ISD}$ - *adequate* test sets.

Class	No. of non-equivalent AMC mutants	Average number of test cases	Average mutation score
c1	1	11	50%
c3	1	9	80%
Average	1	10	65%

Table 12. Mutation score of $\overline{OMD, AMC}$ - *adequate* test sets.

Class	No. of non-equivalent OMD mutants	Average number of test cases	Average mutation score
c6	1	56	100%
c7	5	20	100%
c8	1	42	100%
Average	2.33	39.33	100%

specific constructor was used. For the c3 class, the test that did not kill the ISD mutant did not reach the mutated statement. Since the threshold of 90% coverage was not attained, we conclude that the ISD operator should not be eliminated.

D. PPD

Table 9 shows that the c4 and c5 classes had PPD mutants. However, they turned out to be equivalent mutants. Like the IHD mutation operator, we need to find if there is a system that produces PPD mutants.

E. OMD

The OMD operator deletes overloading methods. Only three classes have OMD mutants, as summarized in Table 9.

Please note that we insisted on eliminating the AMC operator for class mutation testing because it is simulated by the OMD operator. Considering the fact that test cases killing AMC mutants can also kill OMD mutants, we excluded AMC mutants in developing test sets. That is, we used $\overline{OMD, AMC}$ - *adequate* test sets instead $\overline{OMD}$ - *adequate* test sets. Table 12 shows that the $\overline{OMD, AMC}$ - *adequate* tests killed all of the OMD tests. A simple analysis of the OMD mutants reveals that almost any test case that calls the mutated methods would kill the mutants. Therefore, the OMD operator is unnecessary and can be eliminated.

F. EOA

Three classes generated EOA mutants. Among the EOA mutants, one mutant of the c9 class and two mutants of the c11 class turned out to be equivalent. As a result, each of the c9,

Table 13. Mutation score of $\overline{EOA}$ - *adequate* test sets.

Class	No. of non-equivalent EOA mutants	Average number of test cases	Average mutation score
c9	1	13	30%
c10	1	16	90%
c11	1	41	20%
Average	1	23.33	46.67%

c10, and c11 classes has one non-equivalent EOA mutant. The mutation scores for the $\overline{EOA}$ - *adequate* tests are shown in Table 13. The $\overline{EOA}$ - *adequate* tests only killed 30%, 90%, and 20% of the EOA mutants on average, for an overall mutation score of 46.67%. Since the threshold of 90% coverage was not reached, we conclude that the EOA operator should not be eliminated.

4. Summary and Discussion

This section considered the six mutation operators that generated the fewest mutants. Results showed that the MuJava operators AMC and OMD can be eliminated, but ISD and EOA should not.

In the cases of IHD and PPD, no non-equivalent mutant was generated for the 866 classes, so the problem of eliminating those two operators is an open question. However, we note that the classes that produced at least one mutant for the AMC, ISD, PPD, OMD, and EOA, are rather large. This indicates that the object-oriented features which produce few mutants when relevant mutation operators are applied tend to appear only when the class is large. From this observation, we conclude that the IHD and PPD operators should be reexamined with large classes.

In this experiment, we used a threshold of 90% mutation score. However, with so few mutants of the target mutation operators, it may be too high a threshold because one live mutant reduces a mutation score drastically. For example, one live mutant out of a total of five mutants will produce a mutation score of 80%. However, we believe that a higher threshold is still better than using a lower threshold.

V. Related Work

Reducing the number of mutants is one way of reducing the high execution cost of mutation testing. The selective mutation approach [3], [5]-[7] uses a subset of mutation operators that provide almost the same effectiveness as non-selective mutation. However, experiments on selective mutation operators have been conducted mostly with traditional

mutation operators.

After analyzing 22 Fortran mutation operators which generate a large proportion of a program's mutants, Mathur [5] suggested the exclusion of the two most prodigious operators, SVR and ASR. On the other hand, the experiment conducted by Wong [6] used only two mutation operators, ABS and ROR, which were considered the most important. The experiment resulted in an expense reduction of 80% and a size reduction ranging from 40% to 58%.

Offutt and others [7] used the concept of *N-selective mutation*, which omits the *N* most prolific operators. They examined 2-, 4-, and 6-selective mutation, and their experimental trials produced almost the same coverage as non-selective mutation with significant reductions in cost. Extending this approach further, Offutt and others [3] divided the mutation operators into three categories according to the syntactic elements that they modify, namely, statements, operands of expressions, and operators of expressions. Their results show that five mutation operators for the operators within expressions are sufficient for selective mutation.

Barbosa and others [18] indicated that the sufficient operators were completely different for each program suite and defined the *sufficient procedure*, which is a systematic way of selecting a set of sufficient mutation operators.

Smith and Williams [19] empirically investigated the behavior of mutants produced by each mutation operator. They classified mutants into four groups: killed, dead on arrival, crossfire, and stubborn. Their results showed that this categorization can be used in mutation operator selection. However, their case study was conducted against only three Java classes.

VI. Conclusion

This paper described results from an empirical investigation of class mutation operators. Class-level mutants generated by 866 classes drawn from six open-source Java programs were analyzed to characterize the mutation operators. To strengthen the validity of the investigation, the subject applications were chosen to vary in size and application.

According to our results, there were far fewer mutants for object-oriented programs than for procedural programs. In our study, an average of about 37 class mutants and 69 traditional mutants were generated per class. This indicates that mutation testing, which is often considered too expensive for practical unit testing, may be quite practical for object-oriented programs.

Additionally, class mutation operators showed different distributions in the number of mutants. Many class mutation operators produce only a few mutants. For example, the ISI and ISD operators produce fewer than 30 mutants over the 866 classes. This indicates that some object-oriented constructs were rarely used. In particular, there was little use of class inheritance and, hence, polymorphism. However, this does not necessarily mean that the developers were "right" to avoid these constructs. In contrast, the OAC mutation operators produced a very large number of mutants.

One distinguishing feature of class mutation operators is that some of them produced few mutants. This research also conducted a study on the effectiveness of the mutation operators which produce few mutants. Six mutation operators (AMC, IHD, ISD, PPD, OMD, and EOA) were targeted. Among them, only ISD and EOA turned out to generate mutants that other class mutation operators cannot handle. The AMC and OMD operators were identified as not being useful for testing and have been recommended to be eliminated. In particular, the AMC operator showed a strong dependence on the OMD operator. The IHD and PPD operators generated no mutants with our 866 target classes so they may be considered useless. However, more detailed investigation will be needed.

In the future, we plan to extend the effectiveness study so that we can eventually determine a set of selective class mutation operators as was done with traditional mutation operators [3] where prolific operators are eliminated. This will require a systematic evaluation of each operator against a number of classes. We also plan to examine the dependence relation between class mutation operators and hope to use the dependence relation in selective mutation testing. For example, we can say a mutation operator A depends on another mutation operator B if operator A produces mutants whenever operator B does or if operator A produces mutants that can be killed by a mutation-adequate test set for operator B. Therefore, if operator A does not depend on operator B, both operators can be included in a selective mutation operators set.

References

[1] R.A. DeMillo et al., "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, Apr. 1978, pp. 34-41.

[2] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. on Software Engineering*, vol. 3, no. 4, July 1977, pp. 279-290.

[3] A.J. Offutt et al., "An Experimental Determination of Sufficient Mutation Operators," *ACM Trans. on Software Engineering Methodology*, vol. 5, no. 2 Apr. 1996, pp. 99-118.

[4] W.E. Wong and A.P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *The J. of Systems and Software*, vol. 31, no. 3, Dec. 1995, pp. 185-196.

[5] A.P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *Proc. of the 15th Annual Int'l Computer Software and Applications Conf.,* Tokyo, Japan, Sept. 1991, pp. 604-605.

[6] W.E. Wong, *On Mutation and Data Flow*, PhD thesis, Purdue University, West Lafayette, Indiana, Dec. 1993.
[7] A.J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," *Proc. of the 15th Int'l Conf. on Software Engineering*, Baltimore, Maryland, USA, May 1993, pp. 100-107.
[8] R.A. DeMillo et al., "An Extended Overview of the Mothra Software Testing Environment," *Proc. of the 2nd Workshop on SoftwareTesting, Verification, and Analysis*, July 1988, pp. 142-151.
[9] M.E. Delamaro and J.C. Maldonado, "Proteum: A Tool for the Assessment of Test Adequacy for C Programs," *Proc. of the Conf. on Performability in Computing Systems*, July 1996, pp. 75-95.
[10] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur, "Interface Mutation: An Approach to Integration Testing," *IEEE Trans. on Software Engineering*, vol. 27, no. 3, Mar. 2001, pp. 228-247.
[11] S. Kim, J. Clark, and J. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs, *Net. Object Days Conference on Object-Oriented Software Systems*, Oct. 2000.
[12] Y.S. Ma, A.J. Offutt, and Y.R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification, and Reliability*, vol. 15, no. 2, June 2005, pp. 97-133.
[13] Y.S, Ma, A.J. Offutt, and Y.R. Kwon, MuJava home page. online. http://salmosa.kaist.ac.kr/LAB/Mujava/,http://ise.gmu.edu/~offutt/mujava/
[14] A.J. Offutt, Y.S. Ma, and Y.R. Kwon "The Class-Level Mutants of MuJava," *Proc. of the 2006 Int'l Workshop on Automation of Software Test*, pp. 78-84.
[15] R.A. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. on Software Engineering*, vol. 17, no. 9, Sept. 1991, pp. 900-910.
[16] A.J. Offutt and S.D. Lee, "An Empirical Evaluation of Weak Mutation.," *IEEE Trans. on Software Engineering*, vol. 20, no. 5, May 1994, pp. 337-344.
[17] Y.S. Ma, M.J. Harrold, and Y.R. Kwon, "Evaluation of Mutation Testing for Object-Oriented Programs, *Proc. of the 28th Int'l Conf. on Software Engineering*, 2006, pp. 869-872.
[18] E.F. Barbosa, J.C. Maldonado and A.M.R. Vincenzi, "Toward the Determination of Sufficient Mutant Operators for C," *The J. Software Testing, Verification, and Reliability*, vol. 11, 2001, pp. 113-136.
[19] B.H. Smith and LaurieWilliams, "An Empirical Evaluation of Killing Mutants," *Third Workshop on Mutation Analysis (Mutation 2007).*

Yu-Seung Ma received the BS, MS, and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1998, 2000, and 2005, respectively. In February 2005, she joined the Sensor Network Platform Research Team of the Convergence S/W Research Division of Electronics and Telecommunications Research Institute (ETRI), Korea, where she is currently a senior researcher. Her research interests include program testing, mutation testing, and embedded software engineering.

Yong Rae Kwon received the BS and MS degrees in physics from Seoul National University, Korea, in 1969 and 1971, respectively. He received the PhD degree in physics from the University of Pittsburgh, USA, in 1978. He taught at the Korea Military Academy from 1971 until 1974. He was on the technical staff of Computer Sciences Corporation from 1978 until 1983 working on the ground support software systems for NASA/GSFC satellite projects. He joined the faculty of the Department of Computer Science of KAIST, Korea, in 1983. He served as president of Korea Information Society (KISS), chair of KISS SIFSOFT, and steering committee chair of the Asia-Pacific Software Engineering Conference (APSEC). His research interests include verification of real-time parallel software, mutation testing systems, automated test data generation, and software metrics.

Sang-Woon Kim received the BS and MS degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea, in 2000 and 2003, respectively. He is currently a PhD candidate supervised by Yong-Rae Kwon at KAIST. In September 2009, he joined Formalworks, Inc. as chief software developer. His research interests include mutation testing, object-oriented analysis and design, as well as program analysis and software testing in the context of object-oriented development.