Effectiveness of Mutation Testing Techniques: Reducing Mutation Cost

Bouchaib Falah
Al Akhawayn University
33 PUBLICATIONS 132 CITATIONS

Proc. of World Cong. on Multimedia and Computer Science

Effectiveness of Mutation Testing Techniques: Reducing Mutation Cost

[1]Falah Bouchaib, [2]Bouriat Salwa, [3]Achahbar Ouidad
School Of Science & Engineering
Al Akhawayn University in Ifrane
[1]b.falah@aui.ma, [2]salwa.bouriat@aui.ma, [3]o.achahbar@aui.ma

Abstract: **The field of mutation testing has been neglected by industry for a long time because of its high cost. The rising research in this field has resulted in the development of several approaches that aim to reduce the cost of testing and to assess the quality of mutants' generation and destruction. Some approaches suggest decreasing the number of operators, while others intend to reduce the execution time. The goal of this paper is to evaluate the effectiveness of three approaches that reduce the mutation testing cost: class-level, method-level, 10-selective and all operators approach. The experimental part of this research was based on testing seven Java programs using MuClipse tool. The analysis of the resulting findings has proved that the cost of mutation testing can be minimized by selecting a subset of the mutation operators and by adopting the class-level mutation operators for fault detection in java programs.**

Keywords: **Software Engineering; Testing and Debugging; Testing Tools; Mutation Testing**

I. INTRODUCTION

Mutation testing is a fault-based testing technique that measures the effectiveness of test cases. It uses mutation operators that substitute sections of the program to produce slight "syntactic" modifications to the original source code [1]. Therefore, when applying one specific operator, a new version of the program is produced; the resulting version is called "mutant". Each mutant is tested against test suites to assess the effectiveness of test cases in detecting faults [2]. If the tested mutant produces different results than the original program, the tester will induce that the program contains a syntactic error that needs to be corrected. Otherwise, if the tested mutant produces the same expected result as the original program, then test cases have to be improved.

Previous empirical studies have acknowledged the power of mutation testing as a white-box testing technique. Offutt et al. compared data flow with mutation testing to conclude this later is more effective [2]. Moreover, Walsh evaluated mutation testing with statement coverage and branch coverage to conclude mutation testing is again a powerful testing technique [3].

However, the use of mutation testing is not widespread in the industry [4]. Many researchers justify this fact by the high cost of this testing method [5]. Usually, each instruction in the original program can be modified and, therefore, the number of mutants may increase dramatically. In this case, the cost and the time to compile and to test all generated mutants will be very high. According to an experiment that was done by Offut et al., a suite of 10 Fortran-77 programs that include 10 to 48 executable instructions has generated 184 to 3010 mutants [6]. In addition, Mresa and Bottaci have used 11 programs with a mean of 43.7 lines of code,

DOI: 02.WCMCS.2013.1.9

and they have stated that this set of programs have generated 3211 mutants [6]. Hence, mutation testing requires exhaustive execution of test cases that have to be created in order to test all mutants.
Several approaches have been proposed to reduce both the cost and the time of mutation testing. Some approaches proposed to reduce the number of mutants generated and other approaches were designed to optimize the execution process of each created mutant. However, the effectiveness of each technique varies depending on the nature of programs that will be tested as well as the type of mutation operators that will be applied. In this paper, we will focus on the first category which is the reduction of generated mutants. Optimizing the execution process was already investigated by Irene Koo in his paper "Mutation Testing and Three Variations ", where he compared the effectiveness of both weak and strong mutation testing [7]. Thus, the choice of this category would be considered an extension of Koo & al work.
This paper illustrates our quantitative approach in comparing the effectiveness of class-level, method-level, and 10-selective mutation and all operators approach. Class-level approach focuses on testing the main features of object oriented programming languages, such as inheritance and polymorphism. Method-level aims to test the primitive features of programming languages such as replacing, deleting or inserting statements. 10-selective approach is based on selecting the ten most effective operators that were proposed in previous researches [8]. We have focused on 10-selective mutation testing because it was proved to be nearly adequate in a full mutation analysis [9]. Finally, all operators approach is based on selecting both method level and class level operators.
The paper is distributed as follows: section 2 analyzes the previous works done on evaluating mutation testing approaches. It discusses the results of the previous empirical studies to present an updated view of the literature review. Section 3 presents different mutation operators including class-level operators and tradition – or method-level- mutations operators and their uses. A description of the tool and programs used is described in section 4. Finally, last section focuses on analyzing the results of the conducted experiment.

II. Related Works

Several researches have been done in evaluating the effectiveness of different approaches that aim to reduce the cost of mutation testing.
Irene Koo has compared the effectiveness of three different approaches: N-selective mutation, weak mutation and strong mutation testing. Hence, his research adopted the two categories of reducing mutation testing cost: optimizing time execution and reduction number of operators. Irene Koo concluded that selective mutation testing is better than weak mutation testing in terms of mutation score. However, the author has tested C programs with small size, and his conclusion may not apply on large programs [9].
Additionally, Shalini assessed the effectiveness of mutation testing techniques by comparing first order mutant (FOM) and high order mutant (HOM). FOM is generated by seeding one fault a time, while HOM is generated by inserting more than one fault at a time [10]. Shalini concluded that HOMs are harder to kill, which means they are less effective than FOMs. However, HOMs are more efficient because they need less number of test cases than FOMs.
In [11], Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie and Hong Mei1 attempted to compare the effectiveness of selective and random mutation testing. They have based their experiment on C programs that range from 137 to 513 lines. The results of their experiment proved that mutation-selection techniques are not superior to random mutation selection in terms of effectiveness. Moreover, they have concluded that, random selection can achieve very good results when selecting less number of mutants than each operator-based mutant selection technique.

III. Mutation Operators

The efficiency of mutation testing relies heavily on mutation operators used. A mutation operator consists of a set of "predefined program transformation rules" used to substitute a section of the program in order to introduce faults in the source code [12]. The main purpose of mutation operators is to produce different version of the program. The tester produces a set of test cases that compares the result produced by the mutant with the actual result that should be produced. These test cases are executed against the mutants with the intent to produce faulty output. The percentages of mutants killed by the test cases are represented by a mutation score.

Researchers have proposed mutation operators for several languages, including Java. Since this research was based on Java programs, we opted for operators that target object oriented languages. These operators are classified into class-level and method-level operators. Further details about these operators are provided in the next sections.

A. Method-level Operators

The birth of mutation testing can be traced to the late 70’s and early 80’s. At that time, procedural languages dominated the software engineering scene, which influenced the intensive development of mutation operators. This kind of operators is called “traditional operators” or “method-level operators”, and they handle the primitive features of programming languages. Thus, method-level operators modify a subsection of an original program by replacing, deleting or inserting primitive operators (arithmetic operator, relational operator, conditional operator, shift operator, logical operator, and assignment) [13]. Table I represents a set of method-level operators that are used in mutation testing, and which were defined by Offut and Yu-Seung.

TABLE I: METHOD-LEVEL MUTATION OPERATORS [13]

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	Conditonal Operator Delection
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

B. Class-level Operators

Class-level operators were introduced in the late 90’s to address object-oriented programs. OO paradigm introduced several properties such as inheritance, polymorphism, dynamic binding and encapsulation. These new notions introduced different faults in programs that tradition mutation operators did not address [14]. Class-level operators were developed in order to tackle these new types of faults. Table II provides a brief description of the available class-level mutation operators.

TABLE II: CLASS-LEVEL MUTATION OPERATORS [15]

Operator	Description
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	super keyword deletion
IPC	Explicit call of a parent’s constructor deletion
PNC	new method call with child class type
PMD	Instance variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAO	Argument order change

IV. Experiment

A. Program subject

For the conducted experiment, we intended to select program subjects that cover all mutation operators that will be used in the research. We have used 7 Java programs in total, with lengths that varies from 1 to 8 classes. In section 3, we described the traditional mutation operators and categorized accordingly the type of the operators. Thus, we adopted the following categorization to list the features of program subjects:

- Arithmetic mutation operators
- Relational mutation operators
- Conditional Mutation operators
- Logical mutation operators
- Assignment mutation operators

Table III and table IV describe the programs that will be used in the experiment, and denote the presence or absence of different classes of operators we specified earlier. Note that we divided the original table to two tables because of space constraints.

TABLE III: PROGRAM SUBJECT DESCRIPTION – METHOD-LEVEL MUTATION OPERATORS - PART 1

Program Name	Classes Number	Arithmetic Operation	Relational operators
Calculator	1	Yes	No
Student	1	No	No
CoffeMaker	4	No	Yes
CruiseControl	4	Yes	Yes
BlackJack	8	Yes	Yes
Elevator	8	Yes	Yes

TABLE IV: PROGRAM SUBJECT DESCRIPTION – METHOD-LEVEL MUTATION OPERATORS - PART 2

Program Name	Conditio-nal operators	Logical operators	Assig-nment operators
Calculator	No	No	Yes
Student	No	Yes	No
CoffeMaker	Yes	Yes	Yes
CruiseCont-rol	No	Yes	Yes
BlackJack	No	Yes	Yes
Elevator	Yes	Yes	Yes

Table V describes the presence or absence of one or more mutation operators for a specific category.

TABLE V: PROGRAM SUBJECT DESCRIPTION – CLASS-LEVEL MUTATION OPERATORS

Program Name	Class-es Num-ber	Inhe-rit-ance	Poly-morp-hism	Java Speci-fic Feat-ures
Calculator	1	No	No	Yes
Student	1	No	No	Yes
CoffeMa- ker	4	No	No	Yes
CruiseCon-trol	4	No	Yes	Yes
BlackJack	8	No	Yes	Yes
Elevator	8	Yes	No	Yes

B. Automated Mutation Tool

In this experiment, MuClipse was used in order to generate the mutants and compute the mutation score for each test case. Muclipse is a mutation tool and a plug-in for Eclipse and MyEclipse IDE's. This tool was developed based on mμjava, a mutation tool that was developed by Seung, Kwon and Offut [16]. It uses the same mutations operators and mutants' generation process as the previous tool.
The architecture that is adopted by Muclipse is similar to a great extent to the architecture followed by Mμjava. Both implement "Mutant Schemata Generation" (MSG) approach, referred also as "do-faster approach" in the literature [6]. The MSG creates a meta-mutant of every mutant; thus, the compilation requires the compilation of the meta-mutant code and the compilation of the original code, instead of the compilation of the entire set of mutants [8]. This results in a reduced compilation time that characterizes the MSG approach.

C. Approaches

The experiment was conducted on four groups of mutation operators. The mutation operators were divided into four groups. First group, labeled Group1, includes all method-level mutation operators. These mutation operators were used to generate mutants. Second group of mutation operators contains class-level mutation operators. The set of mutants generated by these operators were called "Group2". Third group of mutation operators consists of randomly selected mutation operators. In this case, 10-selective operators were
selected from the set of operators available in MuClipse. The operators are as follow: AOIU, LOI, ASRS, COI, IOP, OMR, JSD, EOA, IOR, PPD. The mutants generated by these operators were labeled "Group3". Fourth group consists of all available mutation operators, both Method-level and class-level operators. The mutants generated by this set of operators were identified as "Group4".
The experiment consists of generating mutants on the program subjects using each group of mutation operators; the resulting mutants are labeled Group1, Group2, Group3, and Group4. The next step consists of running the same unit test cases on each of these mutants, and records the number of mutants that will be killed by each operators group. Finally, the results recorded would be used in order to compare the effectiveness of each group in detecting faults.

D. Results and Analysis

The results of the experiment are represented in table VI and table VII. Table VI represents the number of mutants killed during the test execution for each group of mutants generated. By analyzing this table, it is noticeable that Group 2 performed better in fault detection than Group1. This result has two possible interpretations; the first interpretation would suggest that class-level mutation operators are noticeably more efficient in detecting faults in general. The second interpretation implies that mutation class-level mutation operators generate more mutants; therefore, it makes sense that they kill more mutants during the test. Further research might be needed to investigate this point.
The second observation concerning the number of mutants killed by the test cases concerns Group3 and Group4. The overall observation of the number of mutants killed indicated that mutants of Group3 and Group4 are slightly the same. This means that the results of the use of 43 mutation operators, and the results for choosing 10 random mutation operators are similar. From this interpretation, it is possible to generalize the results and say that 10-selective mutation might be as effective as full-mutation. One should remember that full mutation consists of 43 mutation operators, which means that we have managed to narrow down the mutation operators to 76%. This would improve the cost of mutation testing dramatically.

TABLE VI: MUTANTS KILLED PER PROGRAM AND PER GROUP

Program Name	Group1	Group 2	Group3	Group 4
Calculator	6	32	6	6
Student	0	30	30	30
CoffeMaker	81	42	81	43
CruiseControl	6	59	31	65
BlackJack	7	34	43	43
Elevator	4	39	16	41

Table VII details the mutation score per group and per program. The mutation score in the table is describes in terms of percentage of mutants killed compared to the number of mutants.

TABLE VII: MUTATION SCORE PER PROGRAM AND PER GROUP

Program Name	Group 1	Group2	Group3	Group 4
Calculator	100	71	100	100
Student	0	50	50	48
CoffeMaker	19	17	17	13
CruiseControl	35	26	50	27
BlackJack	64	91	89	89
Elevator	2	7	11	6
Mean	36,67	43,67	52,83	47,17

Figure 1 displays the mutation scores achieved by different groups of the program subject. An analysis of the graph confirms that group3 and group4 mutants perform similarly. In fact, 10-selective random mutation operators perform similar or better than 43 mutation operators based on the mutation score performed by each group. For example, the comparison of the mean of performance of group 3 and group 4 shows that group3 (mean of 52. 83) is performing better than group4 (mean of 47.17).This proves the suggestion saying that the choice of selective mutation testing can reduce considerably the cost of mutation testing without affecting the effectiveness of defect detection.

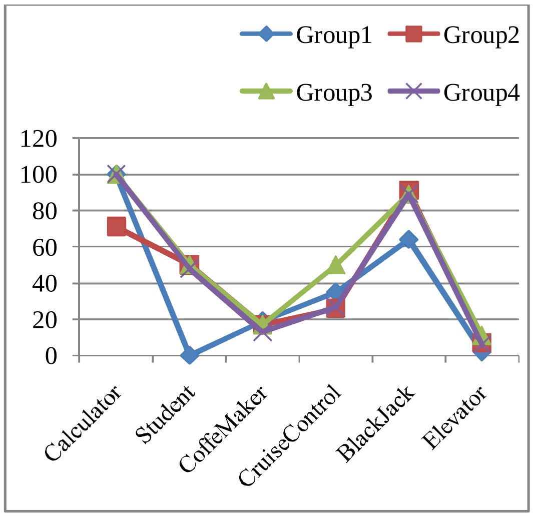

Figure 1: Mutation Scores per Program and per Group

V. CONCLUSION

The results of the research demystified the myth stating that mutation testing is very costly. This research has allowed proving that selecting a subset of the mutation operators can still be as effective as full-mutation. It has also confirmed that class-level mutation operators are more effective than method-level operators in fault detection in Java programs.

This paper opens the opportunity to consider 10-selective mutation testing as a subset of selective mutations. A continuation of the paper could examine all combination of 10 mutation operators, and examine the effectiveness of each combination. This might give more insight about the most effective combination for mutation testing. Furthermore, an enhancement of this research can be done by performing the same experimentsm but on large programs.

REFERENCE

[1] Munawar, H. (2004). Mutation Testing Tool For Java.

[2] A. Jefferson Offutt , Jie Pan , Kanupriya Tewary , Tong Zhang, An experimental evaluation of data flow and mutation testing, Software—Practice & Experience, v.26 n.2, p.165-176, Feb. 1996

[3] Patrick Joseph Walsh, A measure of test case completeness (software, engineering), 1985

[4] Umar. M. An Evaluation of Mutation Operators for Equivalent Mutants. Department of Computer Science King's College, London. 2006

[5] Jeff Offutt , Yu-Seung Ma , Yong-Rae Kwon, An experimental mutation system for Java, ACM SIGSOFT Software Engineering Notes, v.29 n.5, September 2004

[6] Macario Polo , Mario Piattini , Ignacio García-Rodríguez, Decreasing the cost of mutation testing with second-order mutants, Software Testing, Verification & Reliability, v.19 n.2, p.111-131, June 2009

[7] Irene, K. Mutation Testing and Three Variations.

[8] Roland H. Untch , A. Jefferson Offutt , Mary Jean Harrold, Mutation analysis using mutant schemata, Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, p.139-148, June 28-30, 1993

[9] E.S. Mresa and L. Bottaci, "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study," *Software Testing, Verification, and Reliability*, vol. 9, no. 4, pp. 205-232, Dec. 1999.

[10] Shalini,K. Test Case Effectiveness of Higher Order Mutation Testing

[11] Lu Zhang , Shan-Shan Hou , Jun-Jue Hu , Tao Xie , Hong Mei, Is operator-based mutant selection superior to random mutant selection?, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May 01-08, 2010, Cape Town, South Africa

[12] Elfurjani S, Mresa.B. (1999). Efficiency of mutation operators and selective mutation strategies: An empirical study, Volume 9, Issue 4, pages 205–232, December 1999

[13] Yu-Seung, Offut.J. Description of Method-level Mutation Operators for Java, 2005

[14] Yu-Seung Ma , Yong-Rae Kwon , Jeff Offutt, Inter-Class Mutation Operators for Java, Proceedings of the 13th International Symposium on Software Reliability Engineering, p.352, November 12-15, 2002

[15] 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.

[16] The Mutation Process. Retrieved from http://muclipse.sourceforge.net/about.php