Effectiveness of 10-Selective Mutation Testing Technique: Case of Small Programs

Falah Bouchaib, Bouriat Salwa, Achahbar Ouidad
School Of Science and Engineering
Al Akhawayn University in Ifrane
Morocco
{b.falah, salwa.bouriat, o.achahbar}@aui.ma

ABSTRACT: *Mutation Testing is a fault–based software testing technique that 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 10-selective mutation approaches. The experimental part of this research was based on testing seven Java programs using MuClipse tool. The analysis of the findings has proved that the cost of mutation testing can be minimized by selecting a subset (10 operators) from mutation operators.*

Keywords: Software Engineering, Testing and Debugging, Testing Tools, Mutation Testing

Received: 4 June 2013, Revised 8 July 2013, Accepted 14 July 2013

1. Introduction

Mutation testing is a fault-based testing technique that aims at improving 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; if the tested mutant produces different results than the original program, the tester will conclude that the program contains an error that needs to be corrected. Otherwise, if the tested mutant produces the same expected result as the original program, then test cases need 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 stronger testing technique [3]. However, the use of mutation testing is not widespread in the industry [4]. Many researchers justify this fact by the expensiveness of compiling and executing the number of mutants generated [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, 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 [7], where he compared the effectiveness of both weak and strong mutation testing. Thus, the choice of this category would be considered an overlap of Koo & al work.

This research illustrates our quantitative approach in evaluating the effectiveness of 10-selective mutation technique, which is based on selecting 10 mutation operators (both class-level and method-level) [8]. This paper is organized as follows: section 2 analyzes the previous works done on evaluating mutation testing approaches. Section 3 presents different mutation operators including class-level operators and traditional – or method-level- mutations operators and their uses. A description of the tool and programs used in this research is provided in section 4. Finally, last section focuses on analyzing the results of our experiment.

2. Related Work

There are several researches that assess the effectiveness of mutation approaches. These researches have used a variety of number mutation operators, and assessed the influence of these choices on the effectiveness of the testing operations. Dr. Irene Koo has started by evaluating the effectiveness of three approaches: weak mutation, strong mutation, and N-selective mutation. During that process, the researcher has considered two different approaches to improve the cost of mutation testing; the first approaches consisted of reducing the number of operators. The second approach aimed to optimize the execution time of the test cases. The research was done on small size C programs, and mutation scores were used to assess the effectiveness of each approach. The conclusion of the research was that selective mutation testing has better mutation scores than weak mutation [9]. Another researcher done by Zhang, and al. aimed to compare the effectiveness of selective mutation testing versus random mutation testing. The experiment used also small sized C programs as experiment subjects. The experiment concludes that, in terms of effectiveness, selective mutation and random mutation have very similar results [10]. This finding implies that the user can achieve the same result as selective-mutation testing with a smaller number of mutation operators.

3. 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. The main purpose of mutation operators is be to produce different versions 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 percentage of mutants killed by the test cases are represented by a mutation score [11].

Researchers have proposed mutation operators for several languages. 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.

3.1 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) [12]. Table 1 represents a set of method-level operators that are used in mutation testing, and which were defined by Offut and Yu-Seung.

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

Table 1. Method-level Mutation Operators [13]

3.2 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 [13]. Class-level operators were developed in order to tackle these new types of faults. Table 2 provides a brief description of the available class-level mutation operators [14].

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

Table 2. Class-level mutation operators

4. Experiment

4.1 Program subject

For the conducted experiment, we intended to select the program subjects to cover all the mutation operators that will be used in the research. We have used seven small Java programs in total, with lengths that varies from one to eight classes. Table 3 and table 4 describe the traditional operators that we used in each program. Furthermore, both tables list the programs that used in the experiment, and denote the presence or absence of the different classes of operators we specified earlier. (Because f space constraints, we divided table of program subject description into two tables).

Program Name	Classes Number	Arithmetic Operators	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 3. Program subject description – method-level mutation operators - part 1

Program Name	Conditional operators	Logical operators	Assignment operators
Calculator	No	No	Yes
Student	No	Yes	No
CoffeMaker	Yes	Yes	Yes
CruiseControl	No	Yes	Yes
BlackJack	No	Yes	Yes
Elevator	Yes	Yes	Yes

Table 4. Program subject description – method-level mutation operators - part 2

Table 5 describes the presence or absence of one or more mutation operators for a specific category.

Program Name	Classes	Inheritance	Polymorphism	Java specific features
Calculator	1	No	No	Yes
Student	1	No	No	Yes
CoffeMaker	4	No	No	Yes
CruiseControl	4	No	Yes	Yes
BlackJack	8	No	Yes	Yes
Elevator	8	Yes	No	Yes

Table 5. Program Subject Description – class-level mutation operators

4.2 Automated Mutation Tool

We used MuClipse 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.

4.3 Experiment Procedure

The experiment was conducted on two groups of operators'. First group of mutation operators consists of 10-selected mutation operators. In this case, 10 mutation operators were selected from the set of operators available in MuClipse. The selected operators are: AOIU, LOI, ASRS, COI, IOP, OMR, JSD, EOA, IOR and PPD. The mutants generated by these operators were labeled "*Group* 1". The second group consists of all the available mutation operators, both method-level and class-level operators. The mutants generated by this set of operators were labeled "*Group* 2". The experiment consists of generating mutants on the program subjects using each group of mutation operators (Group 1 and Group 2); after generating mutants, next step was about running the same unit test cases on each of these mutants, and then recording 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.

4.4 Results and Analysis

The results of the experiment are listed in table 6 and table 7. Both tables represent the number of mutants killed and the mutation score, respectively, during the test execution for each group of mutants generated. By analyzing both table, it is noticeable that Group 2 performed better in fault detection than Group 1. However, the overall observation of the number of mutants killed indicated that mutants of Group1 and Group 2 are slightly the same. This means that the results of the use of 43 mutation operators, and the results for choosing 10 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%.

Program Name	Group 1	Group 2
Calculator	6	6
Student	30	30
CoffeMaker	81	43
CruiseControl	31	65
BlackJack	43	43
Elevator	16	41

Table 6. Mutants killed per program and per group

Program Name	Group 1	Group 2
Calculator	100	100
Student	50	48
CoffeMaker	17	13
CruiseControl	50	27
BlackJack	89	89
Elevator	11	6
Mean	**52, 83**	**47, 17**

Table 7. Mutation Score per program and per group

Figure 1 displays the mutation scores achieved by different groups of the program subject. An analysis of the graph confirms that group 1 and group 2 mutants perform similarly. In fact, 10-selective 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 both tables shows that group 1 (mean of 52. 83) is performing better than group2 (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.

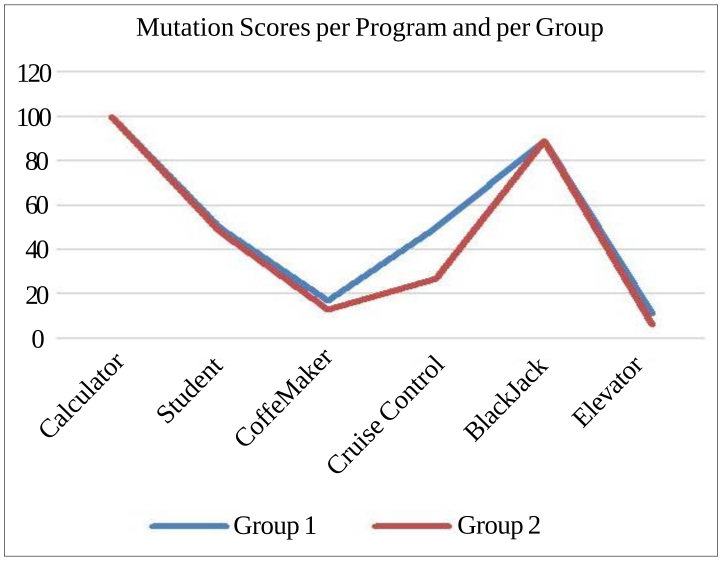

Figure 1. Mutation Scores per Program and per Group

5. Conclusion

The results of the research proved the myth stating that mutation testing is very costly. However, this research has also proved that selecting a subset of the mutation operators can still be as effective as full-mutation.

This paper opens the opportunity to consider all combinations of 10-selective mutation operators and examine the effectiveness of each combination on test programs. Moreover, an extension of this work can involve the same experiment but on large programs and with different categorizations (e.g. logical programs, mathematical programs etc.).

References

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

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

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

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

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

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

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

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

[9] Mresa, E. S., Bottaci, L. (1999). Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study, *Software Testing, Verification, and Reliability,* 9 (4) 205-232, Dec.

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

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

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

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

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

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