RSM: REDUCING MUTATION TESTING COST USING RANDOM SELECTIVE MUTATION TECHNIQUE

[1]Bouchaib Falah, [2]Mohammed Akour, [3]Salwa Bouriat

[1, 3] School of Science & Engineering,Al Akhawayn University, Ifrane, Morocco

[2]Computer Information Systems Department, Yarmouk University, Irbid- Jordan

Email: [1]b.falah@aui.ma, [2]Mohammed.akour@yu.edu.jo, [3]Salwa.bouriat@gmail.com

Abstract

Mutation testing has been neglected by researchers because of the high cost associated with the technique. To manage this issue, researchers have developed cost reduction strategies that aim to reduce the overall cost of mutation, while maintaining the effectiveness and the efficiency of testing. The purpose of this research paper is to present a new cost reduction strategy that cuts the cost of mutation testing through reducing the number of mutation operators used. The experimental part of the paper focuses on the implementation of this strategy on five different java applications. The results of the experiment areused to evaluate the efficiency and quantify the savings of our approach compared to two other existing mutation testing strategies.

Keywords: Testing and Debugging, Mutation Testing, Mutant, Mutation operators

INTRODUCTION

Mutation testing is a powerful fault-based testing technique used to ensure software quality and evaluate the effectiveness of test cases [1, 2, 3]. It is based on seeding faults into the source code by making some syntactic deviations.It uses mutation operators that substitutes sections of the programs to perform slight changes to the original source code [4]. By applying a single mutation operator on the original source code, a new version of the program, called mutant, will be created. Each mutant created will be tested against test cases in order to assess the effectiveness of these tests in detecting the seeded errors [5]. If the mutant produces a result that is different from the original program, the tester can deduce that the program contains a syntactic error that needs to be corrected [6]. In this case, the test suite is said to be "efficient" and the mutant is referred to as "killed" mutant. Otherwise, if the test suite is unable to identify the presence of an error, the mutant is called "alive"mutant.

Unit testing techniques, such as mutation testing, have received lots of criticism due to the large cost associated with them. However, researchers have demonstrated that it is the most effective way to test individual units of software for boundary value [7].Severalempirical studies have proved the strength of mutation testing compared withother white-box testing techniques. Walsh [8], compared mutation testing with statement coverage and branch coverage tests and concluded that mutation testing is the most effective of all three techniques. Moreover, Offutt et al. [5]have compared the effectiveness of mutation testing with data flow andconcludedthat it is the strongest.

Nonetheless, mutation testing has failed to make its industrial debut due to several limitations. Many researchers justify the unpopularity of mutation testing among industrial due tothe high cost associated with the technique [4]. Mutation testing requires large computation resources that necessitate a large storage space and time. The cost of the technique escalates as the scale of the project increase. Howeden [9] has estimated that the number of mutants generated for an n-line program will be of n^2 order. Mutation testing also requires human effortto verify the large number of equivalent mutants and exclude them from the estimates. Equivalent mutants are mutants that are syntactically different than the original program but semantically the same [10]. Offutt et al. [11] have estimated that on average there are about 8.8% of equivalent mutants for each program.

Efforts have been made to remediate the ongoing issue of cost associated with this technique. These efforts have led to the development of numerous strategies with two main focuses. The first set of strategies aim to reduce the cost of mutation by focusing on optimizing specific step during the process of mutation testing such as selective mutation and higher-order mutation. The next set of strategies aims to improve the effectiveness of the technique

by reducing the occurrence of equivalent mutants. Consequently, these mutants cannot be caught, which might flaw the process of mutation.

This paper presents a new cost reduction strategy, called Random Selective Mutation (RSM), which aims to reduce the cost of mutation by reducing the number of mutation operators used during the step ofmutants' creation. The approach operates under the assumption that a smaller number of operators would generate a smaller number of mutants sufficient to perform an effective and efficient testing.

Our approach wastested with five different open source applications downloaded from different internet repositories.The results were compared with the Selective Mutation and Strong Mutation. Strong Mutation performs mutation by applying all operators and testing all mutants [10, 12]. Selective mutation is a do-fewer approach that seeks to reduce the cost of mutation testing by omitting the operators that generates the most number of mutants.

The paper is organizedas follows: section 2 investigates the existing literature on do-fewer approaches. It analyzes previous researches to present an updated view of the literature review. Section 3 presents the mutation operators designed for Java. Section 4 will describe the steps of the suggested algorithms. A description of the tools and the applications used in the experiment is presented in section 5. Finally, section 6 concludes the major findings of this research.

RELATED WORK

Over the last few decades, software engineers have devoted a great effort to find a solution for reducing the cost of mutation testing. Several scholars have studied the process of mutation testing and suggested that it is possible to reduce the cost of mutation testing by following these three approaches: do-fewer, do-smarter, do-faster [10, 13].

Do-fewer approach aims to run fewer mutants without incurring large losses in information. In "do-fewer approach", strategies are used to select subset of the mutants from the set of mutants generated in such way that it is sufficient to assess test cases [10, 12]. Examples of do-fewer approach are selective mutation and mutant sampling.

Do-faster approach aims to generate and run mutants as fast as possible, thought developing a set of cost reduction algorithms [12]. Some do-faster approaches include schema-based mutation analysis and separate compilation.Finally, do-smarter approach tries to distribute computational expenses over several executions [9]. Weak mutation and distributed architectures are good example of "do-smarter" approach.

The cost of mutation testing is often evaluated by the number of mutants used in testing. Selective Mutation is a mutant cost reduction technique that consists of selecting a smaller number of mutation operators to decrease the number of mutants generated [13]. Naim et al. [14] investigated the impact of using a sufficient number of mutation operator on C programs; they concluded that it is not possible to find a unique reduce subset of mutation operators.

Mutant sampling is a do-fewer cost reduction strategy that aims to reduce the number of mutant programs. It consists of randomly selected subset of the mutants generated and executed [12]. Many empirical studies have been conducted to determine the optimal percentage of random mutants to test. A study performed by Bluemke and Kulesza [16] concluded that random sampling around 60% to 50% of mutants reduced the cost of mutation testing while maintaining an adequate mutation score. However, the research could not determine a statistically accurate size of sample that will provide optimal cost reduction and ensure an adequate effectiveness.

Sahinoglu and Spafford [17] suggested a sampling approach by calculating the ratio of the mutant based on the Bayesian sequential probability ratio test. This new approach suggested that the subset of the mutants to be tested is randomly selected until a statistically appropriate sample size is reached.

There are several studiesthat focus on different approaches to speed up mutation testing. Zhan et al. [18] suggested an approach that includes a set of techniques to prioritize and reduce tests. The experimental study proved that it is possible to reduce the execution for all mutants by 50%. The study did not investigate the effect of redundant mutant on the execution time.

Previous researches have demonstrated that the six most used mutation operators generate 40% to 60% of all mutants [19]. Often, test cases that kill other mutation operators have the ability to kill mutants generated from these operators. This new approach suggests a technique to find a reduced subset of mutation operators, such as optimal cost reduction is achieved while maintaining the same effectiveness as full mutation.

MUTATION OPERATORS

The use of mutationtesting is very important in evaluating, comparing, and improving the quality of a test suite. However, the value of mutation testing depends on the set of mutants used in the evaluation. These mutants are formed from the original program through the use of predefined mutation operators. An operator is a rule that substitutes section of the source code in order to perform syntactic modification on the program.

Since mutation is always based on mutation operators, researchers have designed and developed mutation operators to support various programming languages such as Java. The quality of mutation operators is the key to mutation testing.

The main purpose behind the use of mutation operators is to generate many faulty version of the original program [4]. These versions are called mutants. A mutant is created by applying mutation operator on the original source code. An operator is a rule that substitutes section of the source code in order to perform syntactic modification on the program [20]. Fig. 1 illustrates the creation process of mutants.

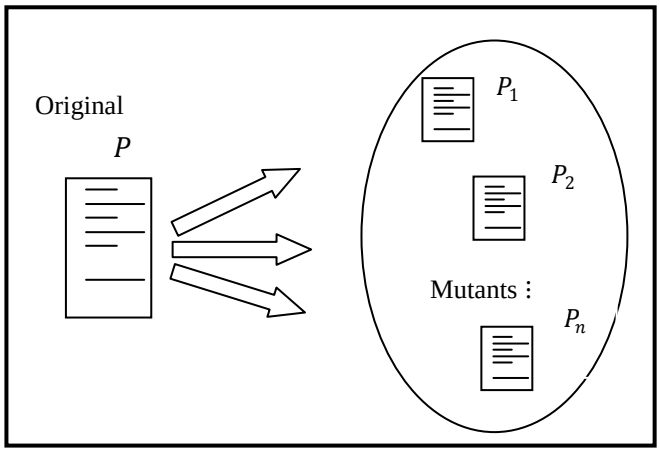

Fig. 1: Mutants Creation Process

The strength of mutation testing resides in the adequate choice of mutants. The test engineer will run the mutants against test cases and compare their behavior with the behavior of the original program with the intention to identifyfaulty results. The percentage of mutants killed by the test cases is referred to as mutation score [20].

Researchers have suggested different mutation operators for each programming language. This research concentrates on Java, where thefocus isonly on the operators that deal with object-oriented feature. The mutation operators supported by Java are divided into two categories: method operators and class operators. The detailsof the mutation operators are presented in the next sections.

3.1 Method-level Operators

Method operatorsalso known as traditional operators were developed based on procedural language features [21]. They perform modifications to statements by inserting, replacing, ordeleting primitive operators [10]. There are six categories of primitive operators:

1. Arithmetic operators,
2. Relational operators,
3. Conditional operators,
4. Shift operators,
5. Logical operators,
6. Assignment

According to the number and type of operands, some of the method level operators are divided into two or three operators. Table 1 describes the method operators defined by Offutt and Yu-seung [21] for Java.

Table 1: Method-level Mutation Operators [21]

Operator Categories	Operator	Description
Arithmetic	AOR	**A**rithmetic **O**perator
	AOI	**A**rithmetic **O**perator **I**nsertion
	AOD	**A**rithmetic **O**perator **D**eletion
Relational	ROR	**R**elational **O**perator
Conditional	COR	**C**onditional **O**perator
	COI	**C**onditional **O**perator **I**nsertion
	COD	**C**onditional **O**perator **D**eletion
Shift	SOR	**S**hift **O**perator **R**eplacement
Logical	LOR	**L**ogical **O**perator **R**eplacement
	LOI	**L**ogical **O**perator **I**nsertion
	LOD	**L**ogical **O**perator **D**eletion
Assignmen	ASR	**A**ssignment **O**perator

3.2 Class Level Operators

Class operators were developed based on object-oriented features [21]. They were introduced in the late 90's to address faults associated with object-oriented features. Object oriented programs differ from traditional programs in many characteristics. They are often structured differently and contain new features such as inheritance, polymorphism, encapsulation, and dynamic binding. Examples of the modification of these features, by the class mutation operators, include deleting the super keyword for inheritance and changing a cast type for the polymorphism.

In Java, class operators are divided into four categories: encapsulation, polymorphism, inheritance, and Java-specific features. These four groups are based on the language features that are affected [22]. The first three categories depend on programming language features that are common in all OO programming languages, while the fourth category depends solely on Java. Table 2 presents the class operators supported by java as well as their descriptions.

Table 2: Class-level mutation operators [21]

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess **m**odifier **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
	IOR	**O**verriding method **r**ename
	ISI	**S**uper keyword **i**nsertion
	ISD	**S**uper keyword **d**eletion
	IPC	Explicit call of a **p**arent's **c**onstructor
Polymorphism	PNC	**n**ew method **c**all 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 **o**perator **i**nsertion
	PCD	Type **c**ast **o**perator **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	**J**ava-supported **d**efault constructor **c**reation
	EOA	Reference assignment and content**a**ssignment replacement
	EOC	Reference comparison and content**c**omparison replacement
	EAM	**A**ccessor**m**ethod change
	EMM	**M**odifier **m**ethod change

THE EXPERIMENT

4.1 Subject Applications

To test our proposedapproach, we have selected five Java applications of different lengths and complexity. The application contained 2 to 9 classes in total. The length of the classes varied from 48 LOC to 347 LOC. Table 3 describes the applications used in our experiment.

Table 3: Description of Subjects of Empirical study

Application	Number of Classes	Number of Methods	LOC
Blackjack	7	465	120
Coffee Maker	4	316	50
Cruise Control	4	544	89
Elevator	8	676	148
Find	1	37	48

In section 3, we categorized the method operators into six categories depending on the type of primitive operator that they manageas follow:

- Arithmetic operators
- Relational operators
- Conditional operators
- Logical operators

- Assignment operators

Table 4 and 5 detail the presence of each type of operator in the subject applications. Note that the original table was divided into two tables in order to answer to space constraints.

Table 4: Program Subject Description – Method-Level Mutation Operators - part 1

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

Table 5: Program Subject Description – Method-Level Mutation Operators - part 2

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

Class operators were distributed into four categories as illustrated by table 6 on the presence or the absence of class operators representative of each category.

Table6: Program Subject Description – class-level mutation operators

Program Name	Number classes	Inheri-tance	Poly-morp-hism	Java specific features
Find	2	No	No	Yes
CoffeMaker	4	No	No	Yes
CruiseCon-trol	4	No	Yes	Yes
BlackJack	8	No	Yes	Yes
Elevator	8	Yes	No	Yes

4.2 Automated Mutation Tool

The creation and execution of mutants are long and resource consuming tasks. Several researchers have worked on developing mutation software that automates these processes. As part of our research, we have used a mutation testing tool, MuClipse [1, 2, 3]. This tool was based on an implementation of the mutation tool MuJava, developed by Offutt [22] and his team.MuClipse is a plug-in for Eclipse IDE.

The architecture of MuClipse is largely similar to the architecture of MuJava. It uses the same mutation operators supported by MuJava. The tool implements a "do-faster" approach using Mutant Schemata Generation technique (MSG) [23]. MSG createsa single-meta-mutant for all of the existing mutants. This requires two compilations: the compilation of the original program and the compilation of the meta-mutant[4]. This results in a reduced time during mutants' creation.

4.3 Proposed Approach

In this paper, we are suggesting a new approach that aims at reducing the cost of mutation testing. The proposed approach works under the assumption that it is possible to reduce the cost of mutation testing by selecting a subset of mutation operators to be used during testing. Instead of generating thousand or tens of thousands of

mutants, we plan to generate a sufficient and reduced set of mutants that would be equally as effective as a full mutation. A smaller number of mutation operators would eventually produce a smaller set of mutants to test. This would reduce the longertime and resources dedicated inexecuting a mutant set.

The selection of mutation operators is performed based on two steps. The first step consists of using a subset of mutation operators while maintaining test effectiveness [12]. To verify this condition, mutation score was used to choose the most effective mutation operators. The second selection consists of randomly selecting a smaller subset of mutation operators based on application size. Since smaller applications usesignificantly less object-oriented features, traditional operators will be privileged over class operators for theseprojects.

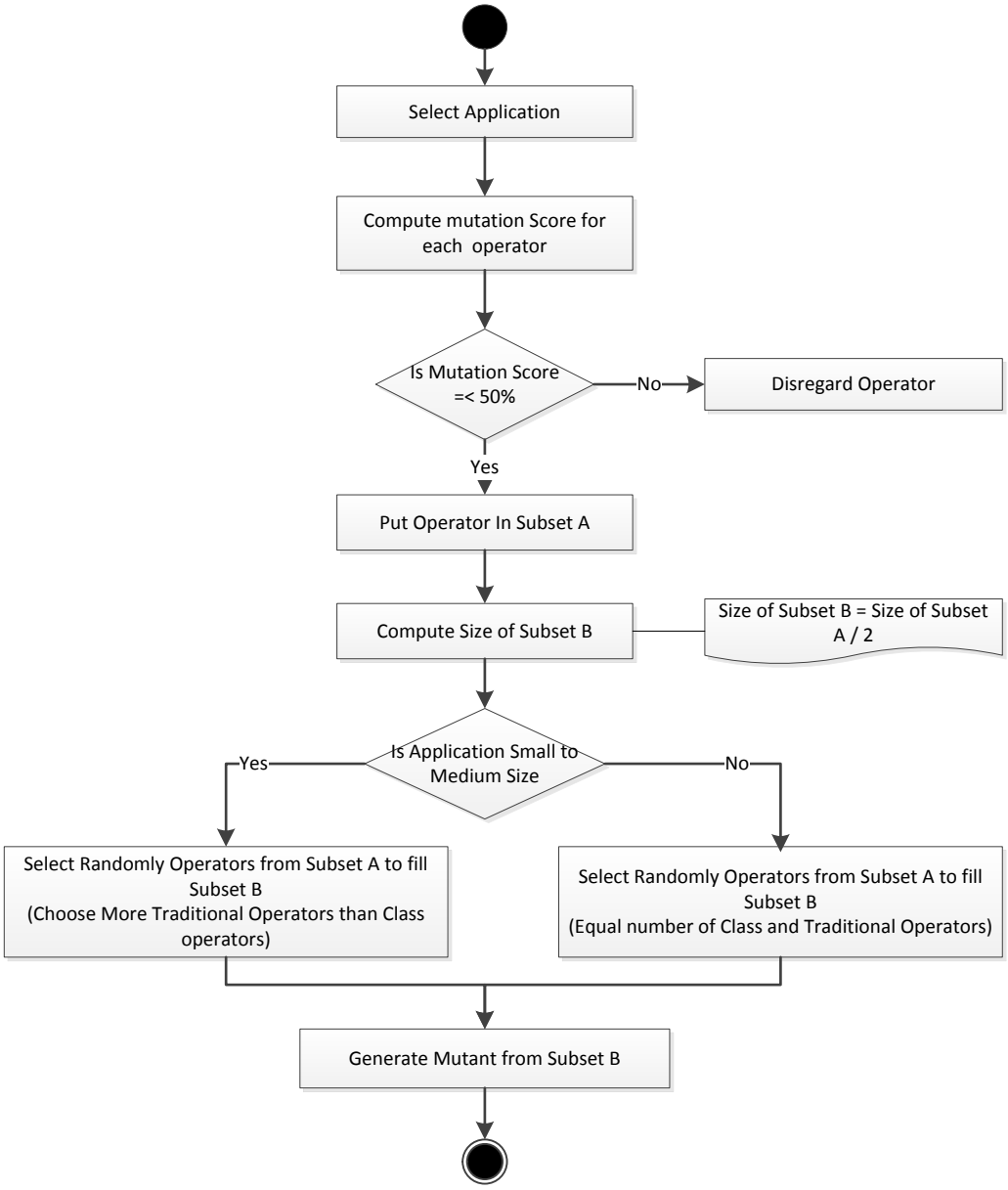

Fig.2:RSM Approach

As shown in Fig. 2, the first step in our approach is selecting a set of applications to be included in our experiments. The mutation score for each mutation operator should be computed before we apply the proposed approach. After measuring the scores, we picked a subset A of mutation operators that have score more than 50% and were not blocked during execution. The equation here is to divide the total number of mutation operators in subset A by 2. The resulting number will be the size of subset B.From subset A, we pickedsubset B on a random fashion. The size of the subset was defined by the previous step.If the application is small to medium size,more method operators are selected than class operators. If its size is larger, equal numbers of method or class operators

are chosen. As a final step, the operators that were selected in subset B on the applications under study programs are applied.

4.4 Results and Analysis

The results of the experiment are represented in Fig.s3 to5. Fig.3depictsthe mutation scores obtained by applying three different strategies, random-selective mutation, and strong mutation, 2-selective and 4-selective mutations. A quick analysis of the graph confirms that our strategy -RSM- achieves similar mutation scores as strong mutation and selective mutation. Note that RSM usesan average of 10 mutation operators, strong mutation used 35 operators, and selective mutation used respectively 33 and 31 mutation operators.

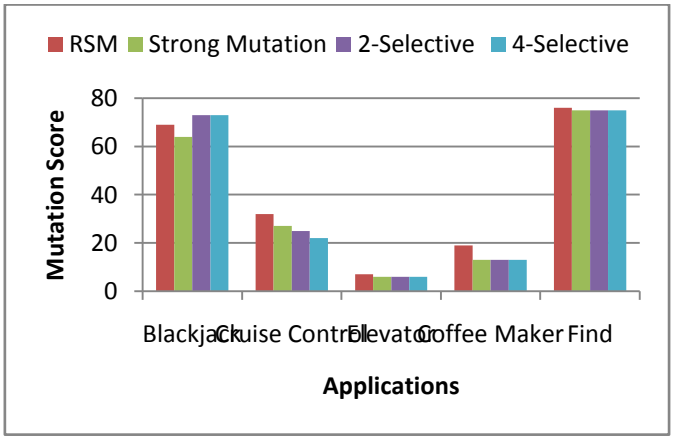

Fig.3**:** Mutation Scores per Application and per Approach

Fig.4 illustrates the test effectiveness scores obtained by each strategy. Aquick analysis of the graph shows that our strategy out-performed the other approaches four out of five times. This confirms that our strategy is more efficient than the two other strategies.

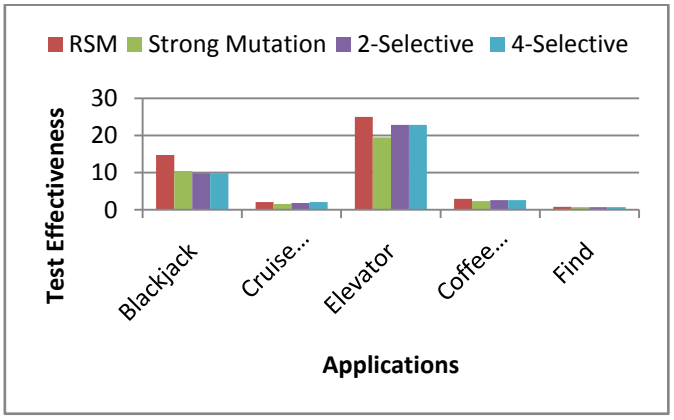

Fig.4: Test effectiveness per Application and per Approach

Random selective mutation approach aims to reduce the cost of mutation, while maintaining similar effectiveness and mutation score as the cost reduction strategies. In order to evaluate the cost saving of our approach, we used the percentage of saving measures. This metric was developed by Offutt et al. [14] to quantify the percentage of mutants that did not have to be generated and killed by a specific cost reduction strategy.

Fig.5 illustrates the percentage of saving of RSM, 2-selective, and 4-selective mutation. It was found that the savings of our strategy were significantly higher than the savings from both 2-selective and 4-selective mutation.

This proves that our approach produces considerably fewer mutants than Selective Mutation without affecting the effectiveness of mutation.

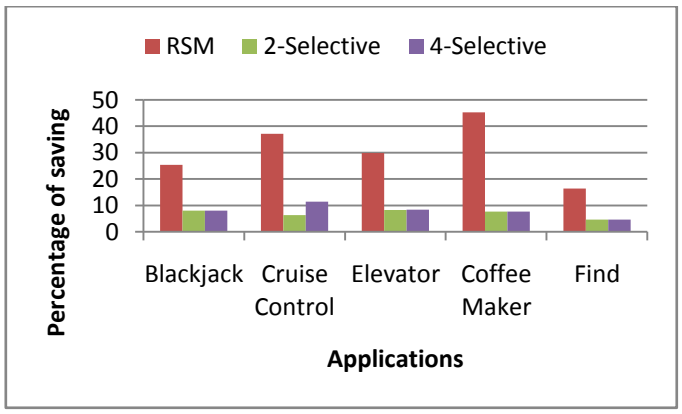

Fig.5: Percentage of Saving per Application and per Approach

CONCLUSION

The results of our experiment have proved that it is possible to perform mutation testing with a small subset of mutation operators. This research has suggested that the choice of the type of mutation operator can have an impact on the effectiveness of mutant detection. It has also confirmed that method operators are more prevalent in smaller sized applications.

This research opens the opportunity to consider a possible relation between the features of the program –such as size - and the type of mutation operators used. A continuation of the paper could examine the effect of size and complexity of application on the type of mutation operator used, and come up with precise formula that would guide the choice of operators. This might give additionalknowledge on the numbers and the types of operator thatprivileged during mutation testing.

REFERENCES

[1] B. Falah, K. Magel. “Test Case Selection Based on a Spectrum of Complexity Metrics”. Proceedings of 2012 on International Conference on Information Technology and Software Engineering (ITSE), Lecture Notes in Electrical Engineering , Volume 212, 2013, pp. 223-235

[2] B. Falah, K. Magel, O. El Ariss. “A Complex Based Regression Test Selection Strategy”, Computer Science & Engineering: An International Journal (CSEIJ), Vol.2, No.5, October 2012

[3] B. Falah. “An Approach to Regression Test Selection Based on Complexity Metrics” , Scholar’s Press,

ISBN-10: 3639518683, ISBN-13: 978-3639518689, Pages: 136, October 28, 2013

[4] Munawar Hafiz, “Mutation Testing Tool for Java”, September 2008, http://www.munawarhafiz.com/research/mutationtesting/MutationTesting.pdf

[5] 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

[6] B. Falah, S. Bouriat, O. Achahbar,” Effectiveness of 10-Selective Mutation Testing Technique: Case of Small Programs,” Progress in Computing Applications (PCA), Vol. 2, Issue 2, September 2013, pp. 73-79

[7] T. Xie, K. Taneja, S. Kale, D. Marinov, “Towards a Framework for Differential Unit Testing of Object-oriented Programs”, AST ’07 Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society Washington, DC, USA, p.5-,2007

[8] Patrick Joseph Walsh. 1985. *A Measure of Test Case Completeness (Software, Engineering)*. Ph.D. Dissertation. State University of New York at Binghamton, Binghamton, NY, USA.

[9] Hong Zhu, Patrick A. V. Hall, and John H. R. May,"Software unit test coverage and adequacy". *ACM Compuer Survey*, Vol. 29, Issue 4, pp. 366-427, December 1997.

[10] Maryam Umar, 2006. An Evaluation of Mutation Operators for Equivalent Mutants, Master Thesis. King's Colledge, London, United Kingdom. Advisor Mark Harman

[11] R. A. DeMilloand A. J.Offutt, "Experimental results from an automatic test casegenerator",ACM Transactions Software Engineering Methodology,Vol. 2, Issue 2, pp. 109 –127, April 1993.

[12] Moohebat, M., Raj, R.G. , Kareem, S.B.A., Thorleuchter, D., "Identifying ISI-indexed articles by their lexical usage: A text analysis approach", Journal of the Association for Information Science and Technology, Vol. 66, No. 3, pp. 501–511. doi: 10.1002/asi.23194.

[13] Eric Wong, "Mutation Testing for The New Century". Advances in Database Systems, Vol. 24. Springer:New York. 978-1-4757-5939-6, 2001.

[14] A. S. Namin and J. H. Andrews, "Finding Sufficient Mutation Operators via Variable Reduction," in Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06). Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 5.

[15] M. Papadakis, N. Malevris, and M. Kintis, "Mutation Testing Strategies: A Collateral Approach," Proceedings of 5th International Conference of Software and Data Technologies, pp. 325-328, June 2011.

[16] I. Bluemke, and K. Kluesza, "Reduction of Computational Cost in Mutation Testing by Sampling Mutants," Proceedings of the 8th International Conference on Dependability and Complex Systems, September 2013, pp. 41-51.

[17] M. Sahinoglu and E. H. Spafford, "A Bayes Sequential Statistical Procedure for Approving Software Products," in Proceedings of the IFIP Conference on Approving Software Products (ASP'90). Garmisch Partenkirchen, Germany: Elsevier Science, September 1990, pp. 43–56.

[18] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, Jul. 2013, pp. 235–245.

[19] A.J. Offutt, and R.H. Untch, "Mutation 2000: Uniting the Orthogonal", Mutation Testing for the New Century, W.E. Wong (Ed.) Kluwer 2001

[20] Elfurjani S, Mresa.B,"Efficiency of mutation operators and selective mutation strategies: An empirical study", Vol. 9, Issue 4, pp. 205–232, December 1999

[21] Y. S. Ma, and J. Offut, "Description of Class-level Mutation Operators for Java," November 29, 2005.

[22] "The Mutation Process". Retrieved from http://muclipse.sourceforge.net/about.php, Access on 2015

[23] Macario Polo , Mario Piattini , Ignacio García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants", Software Testing, Verification & Reliability, vol. 19 Issue 2, pp.111-131, June 2009.