ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

OPKNOT- Schemata Method for Mutation Testing Experimented On Event Driven Web Applications

S. Sumithra[1], T. Ramesh[2]

Research Scholar, Department of Information Technology, Bharathiar University, India, Coimbatore, Tamilnadu, India[1]

Assistant Professor, Department of Information Technology, Bharathiar University, India, Coimbatore, Tamilnadu, India[2]

ABSTRACT: The mutation testing is a testing method aimed at improving the accuracy of the test cases and estimating the number of faults present in system under test. Mutant Clustering takes a subset of mutants using clustering algorithms. Even though the clustering method is effective it also presented with some drawbacks of high cost of undergoing testing. The mutation score result is achieved on two way execution method. Clustering method is to bind the mutants which produces identical results. In this paper a new method for mutation testing to reduce the cost is proposed and it is named as OPKNOT method. OP- mutation operators and KNOT – binding. The OPKNOT method works on mutation operators and on the expression level source code. The mutation operators are bind in an array element and they are executed totally to produce fault code or mutated code into the original code. Then the test cases are executed on the fault code to kill all possible mutant at the same time. By passing the mutation operators into the array style it is confirmed that all possible mutants are created and killed. OPKNOT has the benefits of both mutant creation reduction and execution time reduction method of mutation cost reduction techniques. Mutation testing could be tested on event driven web application to find the effectiveness of the events and event driven tools in a web application.

KEYWORDS: Mutant, event driven, OPKNOT, lexical, semantic, metaprogram, mutation score

I. INTRODUCTION

In case of mutation testing tester mutates (change) certain statements in the source code and check whether the test cases are able to detect the errors. It is a type of white box testing. The changes in mutant programs are kept extremely small, so they will not affect the overall objective of the program. The aim of mutation testing is to assess the equality of the cases which should be robust enough to fail mutant code. This method is called as fault based testing strategy as it creates faults in the program.

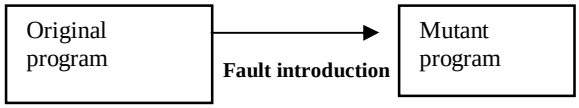


Figure 1.1 Fault Insertion

Mutation testing is a method of inserting bugs into programs to test whether the tests pick them up there by validating or invalidating the tests. Mutation testing was introduced on 1971. It came down due to problems of cost. Mutation testing was researched again recently due to availability of much higher computing power. Most recently mutation testing was used in non-imperative languages such as java and XML. Goals of mutation testing method are to assess the quality of the tests by performing them on mutated code. Use these assessments to help construct more adequate tests. Thereby produce a matching set of valid tests which can be used on real programs. Testing method:

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

Mutant methods are created to take off typical syntactic errors mode by programmers. Many differing mutants are running against the specified tests to assess the quality of the tests. The tests are attributed a score between the original and the mutants. Hypotheses: Mutation testing can only target a subset of all the possible faults that a programmer could possibly make the hope that this is sufficient is based on the following two hypotheses. The component programmer hypotheses: Programmers create programs that are close to be correct. They have the ability and chance to examine their programs in detail. Since a program goes through much iteration most intentions are realized during the design process. Coupling effect: Test data from programs that differ from a correct program by only a small amount is so finely gained that complex errors are implicitly distinguished.

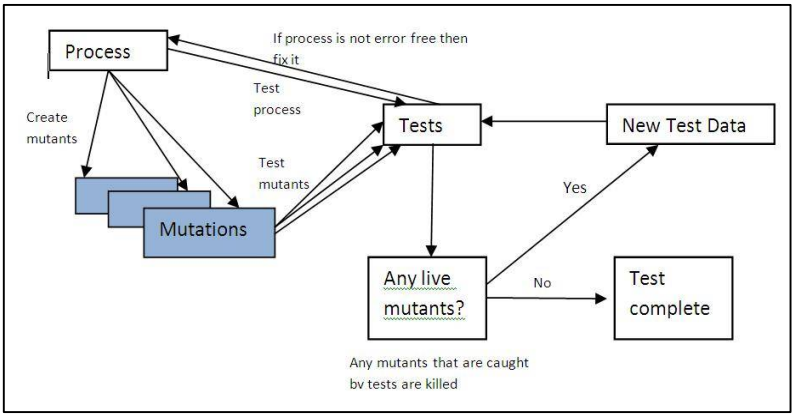


Figure 1.2 Mutation Process

Mutation Process: For a given module M on the test data set D Executing M on test data set D: If outputs are correct, M may have faults and test data is not adequate. Constructing mutants of M: Each mutant is identical to M except for a single syntactic mutation. Executing each mutant on D: if producing a different output, the mutant was dead. If producing a same output, the mutants remains live (D is inadequate; or the mutant is equivalent to M). Terminating mutation testing: Test data set D is adequate. Live mutants are not left or only equivalent mutants left. Otherwise go to the next step. Adding more test cases and repeating: According to information provided by live mutants, design and add more test cases to D, and repeat the above process from beginning. Mutant equivalence: There may be surviving mutants cannot be killed, these are called equivalent mutants. Although syntactically different, these mutants are identical through testing. They therefore have to be checked 'by hand'. Checking through all the equivalent mutants can make mutation testing cost- unaffordable. Even for small programs more human effort is needed to check a large number of mutants for equivalence were almost prohibited. There are few factors that stop mutation testing from being more than an academic topic and being a practical method of testing: The undesirability of equivalent mutants and the cost of checking 'by human effort'. The relatively high computational cost of running all the mutants against a test set. The need for a human effort to verify the contents of output is made more expensive by increases in test cases. This is especially in the case using mutation testing. However methods for limiting the costs involved are continuing to be developed, increasing the changes of industry adoption.

II. PROBLEM DEFINITION

Software testing is an important and critical part of the software development process and confirms the reliability and quality of the software. It also increases confidence in proper functioning and nonfunctional properties of software. About 50% of the cost and time is spent on testing the software thoroughly. For this purpose a number of testing techniques have been developed. One of the problems related to these testing techniques is the generation of effective test data. By solving this problem software testing cost is reduced significantly. As mutation testing is fault based testing technique, it accesses the effectiveness of test data. The main goals of mutation testing are to provide a test adequacy criterion and diagnose faults in the software. It works on the assumptions of competent programmer and coupling effect hypothesis. But the mutation testing suffers from the problem of the huge cost and effort consumption.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

The main reason of this cost consumption is the generation and execution of a large number 35 of mutants. Even a small program might have hundreds of mutants. For example, a program having less than 100 KLOC will generate an enormous number of mutants. The Java quadratic equation program with 25 LOC generated 361 mutants. In this case creating test cases and executing all these test cases on every mutant can be computationally heavy. Each mutant must be executed at least once by the test cases, which leads to too much computation. In case equivalent mutant all the test cases are executed against that mutant in an attempt to kill it. Wong says that the cost can be measured by two metrics, the size of the test set used to fulfill the criterion and the number of the mutants under observation. Based on the existing clustering method, the problem on mutation cost reduction testing is identified. The existing clustering the mutants method is slow, laborious to build and usually unable to completely emulate the intended operational environment of the software being tested. In this paper a mutant schemata method is introduced for solving the problem identified.

III. RELATED WORKS

MUTATION TESTING COST REDUCTION BY CLUSTERING OVERLAPPED MUTANTS, YU-SEUNG MA, SANG-WOON KIM, ELSEVIER (2016)

Although the definition of an overlapped mutant is similar to that of an equivalent mutant, the concept is slightly different. An equivalent mutant is a mutant that is functionally identical to the original program, thus it cannot be killed by any test case. On the other hand, an overlapped mutant is a mutant that is functionally identical to at least one other mutant and can be killed by some test cases if it is not an equivalent mutant. If a mutant is killed (or live), all of its overlapped mutants are killed (or live) in the same manner. Therefore, without executing all mutants, we can predict their results by running only one mutant from each set of overlapped mutants. Consider the statement '$C = A + B;$' and its mutated versions of mutants m_1 and m_2: '$C = A - B;$' and '$C = A + (-B);$' in this example, the mutants m_1 and m_2 are overlapped. Executing both mutants would be a duplicated effort. The identification of overlapped mutants prior to execution would be beneficial; however, the complete detection of overlapped mutants is impossible because it is essentially the same problem as detecting equivalent mutants. Instead, it is focused on a specific type of overlapped mutants, the conditionally overlapped mutant.

MUTANT EXECUTION COST REDUCTION, PEDRO REALES MATEO AND MACARIO POLO USAOLA, (2012)

Another technique used to reduce execution costs is mutant schema, which is based in program schema technique. The program schema technique was proposed by Baruch et al. This technique allows the designer to compose some different programs in the same source code: thus, only one compilation is required to obtain the executable of each program. As all programs are included in a single file, it is necessary to include a mechanism to determine which program included in the schema must be executed, such as a configuration parameter. With mutant schema, the execution environment has to set the configuration properly to run the correct mutant and then, the test cases are executed. Thus the class loading task has to be performed just once in the first execution. This is the reason why some Java mutation tools like Mujava, Javalanche or Bacterio implement mutant schema for mutants generation.

EVENT-DRIVEN WEB APPLICATION TESTING BASED ON MODEL-BASED MUTATION TESTING, ELAHE HABIBI, SEYED-HASSA MIRIAN-HOSSEINABADI, (2016)

This procedure offers a method to tackle the complexity of EDS testing in six stages. During the initial stage, break down the application into different sections by applying the structural division method. In the second stage, graphs will be created from the functionalities of each section. In the following step, the mutated graphs can be inferred from the resulting graphs. To produce test paths in the fourth stage, the coverage criteria will be selected. Longer sequences will be created by merging event sequences in the fifth stage. In the final stage, the derived test cases will be run on the target software, a sample e-mail system, to find the faults. In addition, we evaluate our testing procedure to measure its effectiveness and ability in finding faults through four metrics, i.e. Fault Detection Density (FDD), Fault Detection Effectiveness (FDE), Mutation Score and Unique Fault.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

IV. PROPOSED METHODOLOGY

Mutation technique is very effective technique but still it has some disadvantages such as high cost and high execution time. To overcome this many cost reduction strategies has been developed under two constraints. First one is mutant reduction and second one is execution reduction. The research work is done taking the existing method of clustering the overlapped mutants. Mutant clustering was first proposed in Hussain's thesis chooses a subset of mutants using clustering algorithms. Process starts from generating all first order mutants. A clustering algorithm is applied to classify the first order mutants into different clusters based on the killable test cases. Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in mutation testing and the remaining mutants are discarded. Clustering is the process of organizing objects into groups whose members are similar. Similar between them and dissimilar to other objects belonging to other cluster.

A. *OPKNOT SCHEMATA METHOD*

Unfortunately, current automated mutation analysis system suffers from severe performance problems. Here a new mutant schemata method is been proposed to encode all mutants for a program into one metaprogram. Which is substantially higher than achieved by previous interpretive methods. This method has the advantages of being easier to implement than other reduction technique mainly to reduce the cost of mutation testing two methods have been surveyed, execution reduction and mutant reduction. The proposed mutant schemata follows mutant reduction category along with the benefit of execution reduction method.

B. *DESCRIPTION FOR OPKNOT METHOD*

The proposed mutant schemata method obtains the benefit of both mutant reduction and execution cost reduction. Unlike existing clustering method of mutant, here mutant operators are combined and passed into one single array variable and the whole set of operators are considered as M_1. By executing that single variable, automatically all the mutant operators are executed. Thus the original code is tested with possible fault to obtain the executable of each set of operators.

C. *PARAMETERS PREDICTED FOR THE PROPOSED METHOD*

Parameters are used to validate the proposed method. It evaluates whether the proposed schemata optimizes the existing method limitations. By comparing the results of the parameters the proposed OKNOT method is been evaluated to be a better solution for the existing limitations. The following are some of the possible parameters chosen to compare with other methods.

1. Number of test cases covered during the process.
2. Number of mutants created during the process.
3. Number of Reached mutants among total number of mutants.
4. Total number of killed mutants.
5. Total execution time.
6. Total number of code lines.
7. Mutant score

V. PSEUDO CODE

Step1: Possible mutation operator analysis
Step2: Create array variable for each set of operators
Step3: Pass mutation operators as array elements
Step4: Single execution stores the results of each mutation operator
Step5: Each operator code line contains one single mutant

This technique allows the designer to compose different mutant operators in the same source code. Thus, only one compilation is required to obtain the executable of each mutation operator. As all operators are included in a single

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

array variable, it is necessary to include a mechanism to determine which operator included in the schema must be executed, such as a configuration parameter.

```
// OPKNOT Algorithm as Pseudo code
        // initialize
                Input: Original program, mutated code, Test cases
                Output: Mutation score, Execution time
                M1 – Mutant, V- Variable, EXPR – Expression, MUTk – Mutant Killed, MUTscore – Mutant Score
                MUT t - Total Mutant, MUTe – Equivalent Mutant, EXET – Execution time, STARTT - Start time,
                END T – End time, STARTT ← current time sec

                        Create M array where i=0 to i =M. LENGTH
                        Killed_ mutant ← Mutant operator ← i =0
                // first mutant
                                FOR i=0 to i ≤ M1 . LENGTH
                                        IF (M1 [i] = array element)
                                                EXPR ← V1 (M1 [i]) V2
                                        END IF
                                END FOR
                // second mutant
                                FOR i=0 to i ≤ M2 . LENGTH
                                        IF (M2 [i] = array element)
                                                EXPR ← V1 (M2 [i]) V2
                                        END IF
                                END FOR
                // store the results of mutants killed
                                MUTk ← M1 + M2
                // mutation score
                                MUTscore ← MUTk / MUTt − MUTe
                // total execution time b
                                ENDT ← current time sec
                                EXET ← ENDT − STARTT
                        Return (MUTscore, EXET )
```

Figure 1.3 Algorithm as Pseudo Code

VI. SIMULATION ENVIRONMENT

With the existing simulation method a testing environment is created. Eclipse Europa version 6.0 is been chosen as java simulator. Eclipse Europa IDE is specific for java and J2EE languages. For web application testing J2EE API is used. For running the test case for event and evet driven tools from the web application, it should be run on local-host environment. Tomcat-server is configured as local-host server. Java CC is Java C- Compiler plugin and it is used to compile the source code of E-Compiler project. Using JDBC-ODBC connectivity E-Compiler has been connected to SQL server database. E-Compiler is separated as six modules. After executing the E-Compiler, the source code should be transformed to mutated code for doing mutation testing process. Each module is translated to mutated code. For creating mutants MuClipse plugin is inserted into the eclipse environment. ExtendedOj is another plugin which supports MuClipse. After mutating (fault code) the original code the test cases are executed on the mutated code. For creating test cases JUnit test case plugin is inserted into eclipse environment. Mutation testing is used for evaluating the test cases developed over the source code. So for creating test cases JUnit test case plugin is used. Before executing the test case the total number of mutants created is noted. By executing the test cases over the mutated code the result will be evaluated. When the existing method results are obtained, the proposed OPKNOT method is implemented into the

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

source code as in the same procedure. The OPKNOT method works on the expressions in the source code. OPKNOT schemata method binds the operators and mutants is created in the source code. Total number of mutants created is noted. The execution time is also noted for both existing method and proposed method. By comparing two results the improvisation and betterment of the proposed method is realized.

VII. PERFORMANCE EVALUATION

A. *E-COMPILER WEB APPLICATION TESTING*

The main objective of the project is to visualize the things that goes under each and every phase of a compiler. With the help of this product the users can easily understand the compiler. In this project it visualizes the java programs. In the first phase, it visualizes the tokens that are identified from the uploaded java program done by the user. In the second phase, it generates the parse trees both in top-down and bottom-up view based on the tokens. In the third phase, it shows the semantic view that concentrates on syntax checking and type-casting operation. In the fourth phase, it generates the intermediate code based on the syntax trees. In the fifth phase, it removes the unnecessary temporary variables from the intermediate code. Finally, the output will be displayed for the given program.

B. *MUTATION REDUCTION RATIO*

Figure 1.3 shows the result obtained on mutation reduction ratio. The main goal of OPKNOT method is to reduce the mutants created and executed. Thus on analyzing the results and the graph it is well understood that OPKNOT method reduces the number of mutants.

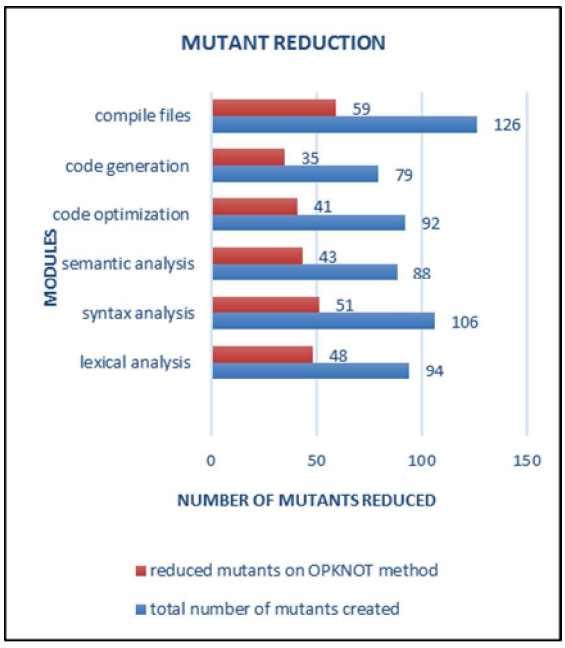


Figure 1.3 Graph of mutation reduction

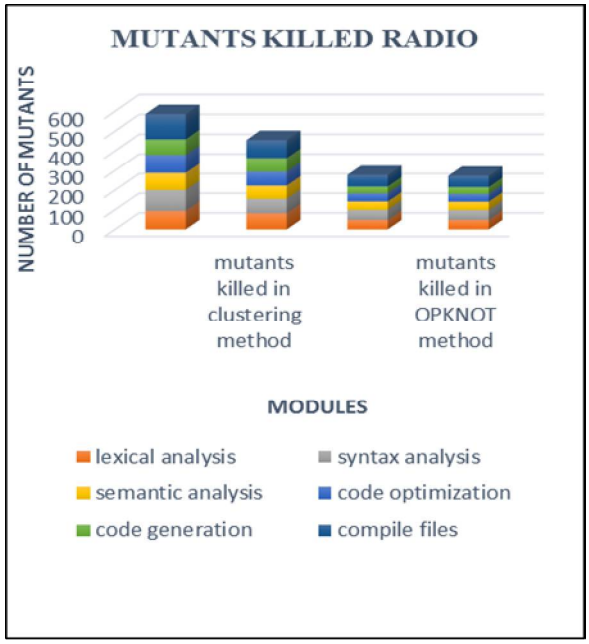


Figure 1.4 Mutant Killing Ratio

C. *MUTANT KILLING RATIO*

Figure 1.4 shows the mutant killing ratio on comparing existing clustering method and OPKNOT method. Another aim of the proposed OPKNOT method should reduce number mutant creation and increase the mutant killing ratio. On analyzing the graph on figure 1.4 it is clear that OPKNOT method kills the maximum number of mutants.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

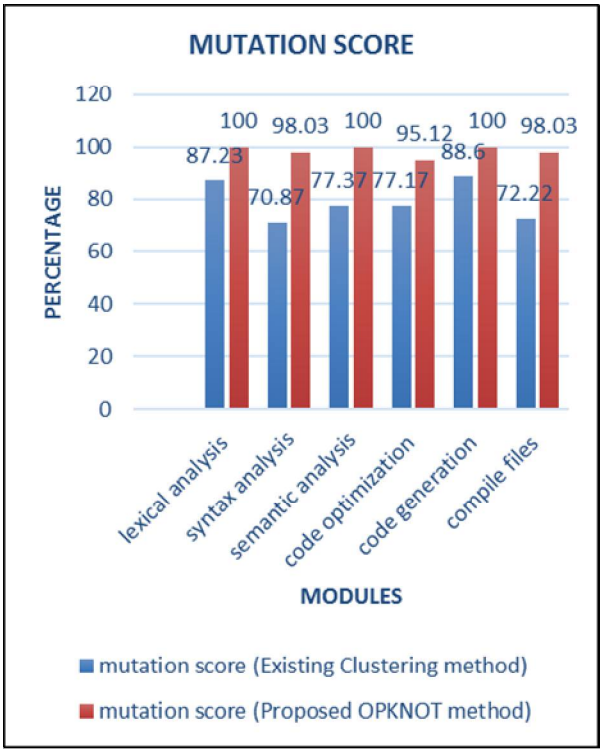
(An ISO 3297: 2007 Certified Organization)

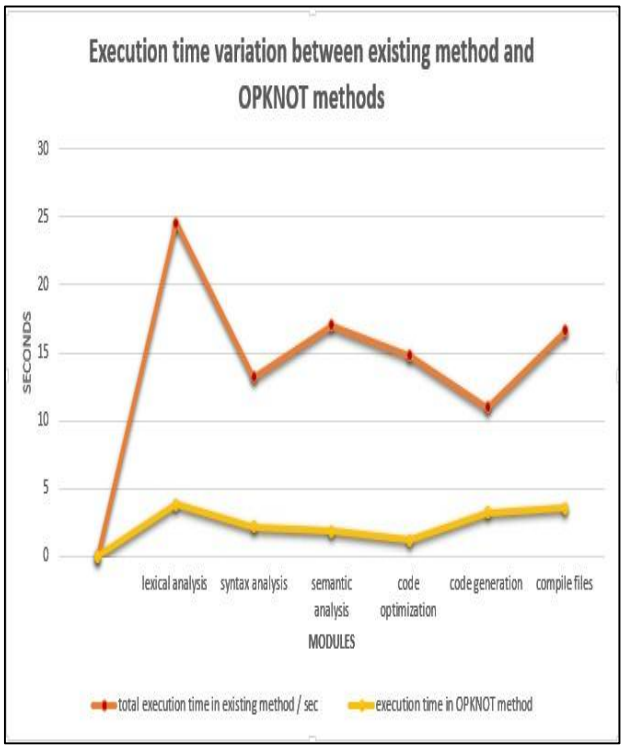
Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

D. ***MUTATION SCORE***

Figure 1.5 shows the mutation score obtained on applying the OPKNOT method on the web application modules. Mutant score is that the through put of the proposed OPKNOT method of number of mutant killed on total number of mutants created. Therefore by analyzing the graph in figure 1.5 it is understood that OPKNOT method maximizes the mutant score to 100% while existing clustering method gives less than 85%.





E. ***EXECUTION TIME REDUCTION RATION***

The execution time reduction ratio is showed in figure1.6 after applying the proposed OPKNOT method on the web application. The efficiency of the proposed OPKNOT method is proved by calculating the mutation score and the execution time. By analyzing the graph in figure 1.6 it is well understood that OPKNOT method has great efficiency and improvised when compared to existing clustering method.

By analyzing the results obtained the difference between existing and OPKNOT method is verified. It is noticed that each parameter values shows how the existing method differs from the proposed method. Mutant creation is reduced in proposed method. Mutant killing ratio is increased and mutation score also increased. Thus the overall execution time is automatically reduced. This reduces the cost of the web application mutation testing. The graphical representation shows the results clearly that could be easily understood. OPKNOT method executes only one mutant that holds the whole mutation operator set. It make sure that all possible mutants are executed. Only one execution is needed to execute all the mutants. At the end of the execution the results of each operator are stored in separate variable. Unlike mutant clustering method. It is efficient for more test cases. All mutant execution is found by comparing the number of array elements and the results. Thus the execution cost as well as the number of mutant creations could be reduced.

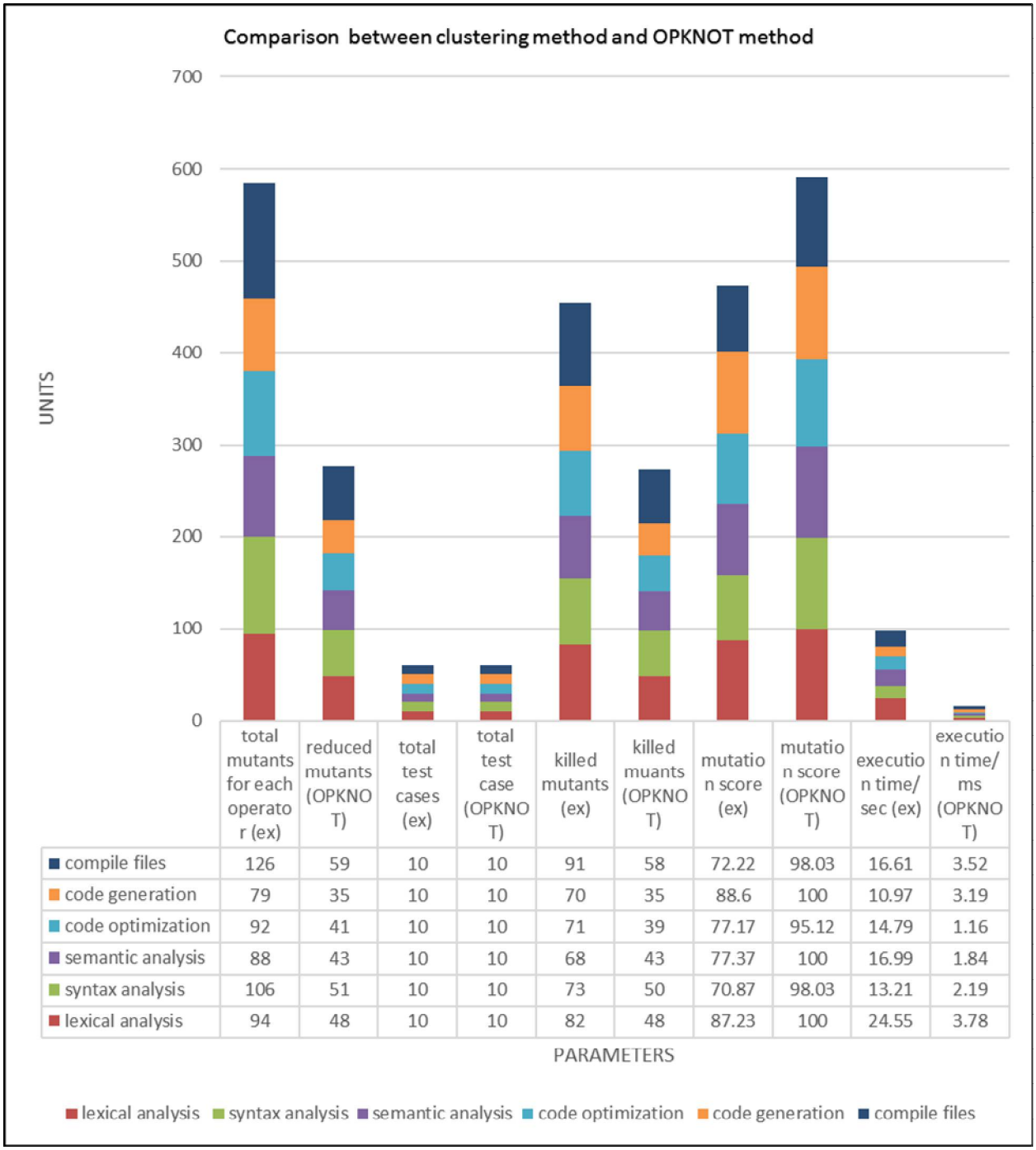
ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017



	total mutants for each operator (ex)	reduced mutants (OPKNOT)	total test cases (ex)	total test case (OPKNOT)	killed mutants (ex)	killed muants (OPKNOT)	mutation score (ex)	mutation score (OPKNOT)	execution time/ sec (ex)	execution time/ ms (OPKNOT)
compile files	126	59	10	10	91	58	72.22	98.03	16.61	3.52
code generation	79	35	10	10	70	35	88.6	100	10.97	3.19
code optimization	92	41	10	10	71	39	77.17	95.12	14.79	1.16
semantic analysis	88	43	10	10	68	43	77.37	100	16.99	1.84
syntax analysis	106	51	10	10	73	50	70.87	98.03	13.21	2.19
lexical analysis	94	48	10	10	82	48	87.23	100	24.55	3.78

Figure 1.7 Overall results between clustering and OPKNOT methods.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

VIII. CONCLUSION AND FUTURE WORK

Mutation testing is a fault based testing technique that works in conjunction with traditional testing techniques. It is done by generating faults and observes the effect by going through three conditions: reachability, infection and propagation. Mutation testing suffers from the problem of large number of mutant creation and large effort, time and cost consumption. The proposed approach is inexpensive for mutation testing. It works on the reduction of mutation cost as well as execution time. OPKNOT schemata method handles different parameters to show that it has been improvised from existing clustering method of mutation testing.

On analysing the overall results of existing method and proposed OPKNOT method, each parameter is evaluated for each module in the sample E-compiler project. The number of mutants created for existing method were 585 for each mutation operator. After using OPKNOT schemata 277 mutants have being created. 47.35% of mutant creation has been reduced after using OPKNOT method. In existing method 455 mutants have been killed out of 585 mutants while using OPKNOT method 273 mutants have been killed out of 277 mutants. Therefore killing ratio has been increased in OPKNOT method. The improvisation is proved by analysing the mutation score between existing and proposed method. The overall mutation score for each module in existing method is 78.91% where the overall mutation score for OPKNOT method is 98.53%. The execution time causes the cost consumption of mutation testing. So the execution time is compared between existing method and OPKNOT method. The execution time consumed in existing method is 97.12 seconds where execution time consumed in OPKNOT method is 15.68 seconds. These parameter results have been analysed using the table of results and the graphical representation. From these analysis it is proven that the proposed OPKNOT method reduces the mutation testing cost by reducing the mutant creation and time consumption.

In future the researchers could be able to rectify the drawbacks of the proposed method. OPKNOT method is applicable only for expression level mutants. But there are N number of mutation operators for each line of code. OPKNOT method could be improved on other levels of mutation testing. The proposed method is under limited test cases. In future it could be enhanced to more test cases. The OPKNOT method could be created as a .jar file or a plugin file that could be used a template in eclipse. The proposed method developed on Java language. In future OPKNOT could be developed on other languages. Mutation testing is carried out on various languages and various projects. The proposed OPKNOT method could be enhanced to support various platforms and languages.

REFERENCES

1. C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis," in Proc. 21st International conference on Software Engineering and Knowledge Engineering (SEKE), 2009, pp. 422–425.
2. R. A. De Millo, D. S. Guindi W. McCracken, A. Offutt, and K. Ruler. A widened chart of the Mutation programming testing environment. In Software Testing, Verification, and Analysis, 1988. Procedures of the Second Workshop on, pages 142-151. IEEE, 1988.
3. J.J. Dominguez- Jiimenez, A.Estero- Botaro , A. Garcia – Dominguez, I. Medina- Bulo, 2011, "Evolutionary Mutation Testing", 0950 – 5849, Elsevier – Information and Software Technology, doi: 10.1016/j. infsof.2011.03.008.
4. Fathima M M.E, Uthra V, 2015, "Analysis of Scrum based mutation testing for safety critical projects", International Journal of Advanced Research in Computer Science and Software Engineering 5(9), September- 2015, pp. 524-528.
5. M. Gligoric, L. Z. C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in Proc. Intl. Symp. Softw. Testing and Analysis (ISSTA). ACM, 2013, pp. 224–234.
6. Harrold M.J., Kim S.W., and Kwon Y.R., ―MUGAMMA: 2006, Mutation Analysis of Deployed Software to Increase Confidence Assist Evolution‖, Proc. Second Workshop Mutation Analysis, pp. 10, 2006.
7. Ignacio Garcia-Rodriguez, Macario Polo and Mario Piattini, 2009, "Decreasing the cost of mutation testing with second-order mutants", Software testing verification & validation, 2009; 19:111-131, doi: 10.1002/Stvr.392.
8. A. Jefferson Offutt, A practical system for mutation testing: help for the common programmer, in: Proceedings of the International Test Conference, 1994, IEEE, 1994, pp. 824–830.
9. Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering.37 (5): p. 649-678, September 2011, 2011.
10. JUnit. JUnit, Testing Resources for Extreme Programming. 2003 [Access January 5, 2003]; Available from: http://www.junit.org.
11. Mark Harman, Yue Jia, 2009, "Higher Order Mutation Testing", 0950-5849, Elsevier – Information and Software Technology,doi:10.1016/j.infsof.2009.04.016.
12. S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in Proc. International Conference on Software Testing, Verification and Validation (ICST). IEEE Computer Society, 2013, pp. 74–83.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

13. A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in Proc. International Conference on Software Engineering (ICSE). ACM, 2008, pp. 351–360.
14. M. Polo, M. Piattini and I. García-Rodríguez. "Decreasing the cost of mutation testing with 2-order mutants." Software Testing, Verification and Reliability. 19(2): p. 111-131, 2008.
15. P. Reales, M. Polo and J. Offutt. "Mutation at System and Functional Levels." In Third International Conference on Software Testing, Verification, and Validation Workshops. 110-119, Paris, France, April, 2010.
16. Steffen Herbold, Jens Grabowski, Stephan Waack, A model for usage-based testing of event-driven software, in: 5th International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C), 2011, IEEE, 2011, pp. 172–178.
17. W. Wong and A. P.Mathur. Decreasing the cost of Mutation testing: A test study. Journal of Systems and Software, 31(3):185 - 196, 1995.
18. L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Head based and subjective mutant determination: Better together. In IEEE/ACM International Conference on Automated Software Engineering. ACM, 2013.In IEEE/ACM International Conference on Automated Software Engineering. ACM, 2013.