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

A Review on Different Approaches of Mutation Cost Reduction Techniques

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: A mutation is a small change in a program. Such small changes are proposed to model low level defects that arise during the process of coding a software system. Mutation testing is a structural testing method designed for assessing and improving the accuracy of test suites and identifying the number of faults present in the software code under test. Mutation testing is carried out after the failure of traditional structure coverage analysis and manual peer review has failed. The main drawback in mutation testing is that the cost of executing the mutants is high. But researchers have provided many cost reduction techniques to be applied during mutation testing. This paper provides a comprehensive view of all the cost reduction techniques developed and tested. This paper also presents the recent trendy experimental results on mutation testing. The literature review part covers all the recent developments of mutation testing.

KEYWORDS: Mutation, clustering, mutant schemata, first order mutant, second order mutant, high order mutant.

I. INTRODUCTION

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, equally, that it does not do anything unintended. Software should be predictable and consistent, presenting no surprise to users. In this research, many approaches have been explained to achieving this goal. Mutation testing is used to assess the quality of the test set on the bases of its capability to reveal simple faults injected in the program being tested. The goal is to achieve high mutation score. To achieve high mutation score, test set should be able to kill all mutants, but some however syntactically different from the original program. These mutants are contributed to increasing the computation cost.

II. RELATE WORK

Mutation testing has been actively investigated by researchers since the 1970s and remarkable benefits have been achieved in its theory, concepts, technology and empirical evidence. While the latest realizations have been summarized by existing literature review [12], the lack insight into how mutation testing is actually applied. The goal is to identify and classify the main applications of mutation testing and analyse the level of replicability of empirical studies related to mutation testing. This section provides a systematic literature review on the application perspective of mutation testing based on a collection of survey papers published. In particular, by analysing the testing activities mutation testing is used, which mutation tools and which mutation operators are employed. The results show that most studies use mutate on testing as an assessment tool aimed for unit tests, and many of the supporting techniques for making mutation testing applicable in practice are still under development. Based on the observations made, nine recommendations for the future work, in addition to an important suggestion on how to report mutation testing in testing experiments in an appropriate manner.

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

III. TECHNIQUES

A number of techniques have been proposed for reducing cost consumption and finding equivalent mutants. Some of these techniques are implemented by using some algorithms. Mutation testing requires number of test set for testing of code, but these test sets may not be good for revealing faults in the program. Cost consideration may requires the generation of only one adequate test set with respect to a given code. The cost of the mutation testing is the major bottleneck in its use by practitioners. To overcome the problem of cost, two alternatives are used in [01] which distinguishes equivalent and non-equivalent. The randomly selected x% mutation criterion: In this mechanism randomly select the x% of each type of mutant and ignore the remaining mutants. The percentage of selected mutants varies between 10% and 100% and the keeps on increasing with constant value to examine the cost effectiveness of mutation testing using a different percentage of mutants. The constrained mutation criterion: Another mechanism examines only a few specific types of mutants and ignores the others. During execution of mutation testing using above mentioned criteria, those test cases are discarded which are note able to distinguish at least one non-equivalent mutant. A good selection of mutation operators can reduce the examination cost dramatically without sacrificing the fault detecting capability of mutation testing. During execution of mutation testing using above mentioned criteria, those test cases are discarded which are not able to distinguish at least one non-equivalent mutant. This is a good selection of the fault detecting capability of mutation testing.

A. *EVALUATION OF SELECTIVE MUTATION TESTING WITH SECOND ORDER MUTANTS*

Marinos Kintis [02] surveyed here are the selective mutation based. It is a way to save execution cost by reducing the number of mutants that need to be executed. They show the effectiveness of selective mutation testing over non-selective mutation testing. As the major execution cost of mutation testing is incurred when all the mutant programs runs against the test cases. The number of mutants generated for a program is roughly proportional to the product of the number of data references and the number of data objects. To perform selective mutation testing, firstly create the selective mutants for program to test. Then kill the mutants as much as possible and measure the effectiveness of results. In selective mutants, efficient operators are used for the generation of mutants. Only those mutation operators are expected, which generate mutants that require the construction of test cases. Those test cases are able to kill not only the mutants of the given operator but also mutants of other operators at a low cost are called effective test cases. Goal of selective mutation is the use of operators that tend to produce mutants having semantically small faults. These results signify that the use of efficient operators can provide efficiency for selective mutation. It has almost the same coverage as that of non-selective mutation with significant reduction cost.

B. *DECREASING THE COST OF MUTATION TESTING WITH SECOND ORDER MUTANTS*

Macario Polo [03] surveyed here has presented a technique to decrease the cost of mutation testing by the combination of first-order mutants. The idea is based on the reduction in the number of mutants by combining mutant pairs and generating into new mutants. Each mutant is created with the introduction of two faults in the mutants, from where they come. Results lead to believe that mutant’s combination does not decrease the quality of the test suite, whereas it assumes important saving s in mutant execution and result analysis. Three combination strategies have been implemented by the testooj tools are LastToFirst, DifferentOperators and RandomMix. Thus reducing the number of mutants reduced to half of the original suite implies a significant reduction in the effort devoted to mutant execution and result analysis. **Types Of Mutant as follows:**

- **Killed mutant:** On execution mutant produces different result from the original program.
- **Alive/Live mutant:** Test cases are not able to detect the injected fault in the program.
- **Equivalent mutant**: A mutant that always produces identical results with the original program.

C. *CONDITIONS FOR TEST DATA TO KILL A MUTANT*

Three conditions are needed to satisfy test data to kill a mutant. If P denotes a program, M denotes a mutant of P on statement S and T denotes the test data for P. The conditions are as follows:

- Reachability condition: S must be reached, because a mutant is presented as a syntactic change only in statement S and the other statement in mutant are syntactically equivalent to original program. If I cannot reach the statement S, impossible to kill M.

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

- Necessity condition: T must be able to bring M to a different state than that of P on statement S. For T to kill M, it is necessary S is reachable and on execution the state of M must be different after execution of S. because the rest of statement of P and M are syntactically equivalent.
- Sufficient condition: The final state of M must be different from P. The different state caused by the necessity condition should propagate through the program's computation to result in a different output.

D. *EVOLUTIONARY MUTATION TESTING AND HIGHER ORDER MUTATION TESTING*

Evolutionary Mutation Testing (EMT) [04], which generate and execute only some of all the mutants, while preserving testing effectiveness. EMT generates and selects mutants in a single step, reducing the number of mutants to execute by preferring through the fitness function two kinds of mutants: surviving mutants (which have not been killed by the test suite) and difficult to kill mutants (which have been killed by one and only one test case that kills no other mutant). Evolutionary Mutation Testing (EMT), which uses an evolutionary algorithm. This technique generates mutants on demand, as required by the selection process. It reduces the number of mutants by favouring potentially equivalent mutants (equivalent and stubborn non-equivalent mutants) and difficult to kill mutants. On considering these to be strong mutants. Strong mutants are used to improve the quality of the initial test suite. Here introducing the concept of subsuming Higher Order Mutants (HOMs) [05], one that is harder to kill Higher Order Mutant (HOM) than the first order mutants (FOMs) from which it is constructed. Subsuming HOMs denotes the subtle fault combinations. It is preferable to replace the FOMs with the single HOM. It also removes the number of myths related to the mutation testing. A strongly subsuming HOM is only killed by a subset of the intersection of test cases that kill each First Order Mutant from which it is constructed. The set of test cases that kill HOM also kill each and every FOM. Fewer (but better) mutants mean requirement of fewer (but better) test cases. The search based approach for the identification of efficient subsuming HOMs is told. This approach also overcomes the exponential explosion in the number of HOMs. The algorithm targets subsuming up all HOMs rather than searching for strongly subsuming HOMs. It uses greedy algorithm, a genetic algorithm and a hill climbing algorithm. The result indicates that the performance of genetic algorithm is best among all of them. However, others also improve the quality of results. The higher order mutation testing may turn out to be far more scalable than first order mutation testing.

E. *SEMANTIC MUTATION TESTING AND MEMORY MUTATION TESTING*

Semantic mutation testing [06] mutates the semantic in which the description is written. The mutated semantics of the language represent possible misunderstandings of the description language and thus capture a different class of faults. Since the misunderstandings are extremely context dependent, this context should be used to determine which semantic mutants should be produced. Semantic Mutation Testing is a powerful and general concept. It requires the use of a description language with a semantics that allows manipulation along with some notion of likely misunderstandings. Instead, the mutations of the semantics might discover possible variance in the semantics. Example, many programming languages have elements that are implementation specific, the compiler writer is allowed to choose between certain options. Memory Mutation Testing [07], with a more extensive and precise analysis of 18 open source programs, including two large real-world programs, all of which come with well-designed unit test suites. Specifically, the empirical study makes use of recent results on Trivial Compiler Equivalence (TCE) to identify both equivalent and duplicate mutants. Though the literature on mutation testing has formerly deployed various methods to cater for equivalent mutants, no previous study has provided for duplicate mutants.

F. *PATH AWARE MUTATION REDUCTION AND SCRUM BASED MUTATION TESTING*

A path-aware approach to mutant reduction [08], which explores mutant reduction from the outlook of the path depth in the program under test. A program fault is connected with various characteristics, such as fault location, fault type, etc. These fault characteristics, affect the behaviours of a program. The multiplicity among selected mutants is achieved by considering different experiential rules. Scrum model and mutation testing [09] are compatible to generate test cases in unexpected real time scenarios for better performance. Therefore Scrum based mutation testing can be used for safety critical projects. Mutation testing combines the strategy of Scrum along with mutation testing for its efficient use in building safety critical projects.

G. *MUTATION ANALYSIS USING MUTANT SCHEMATA*

A. Jefferson Offutt [10] performs mutation analysis with the use of program schemata to encode all mutants of a program into one metaprogram, which is compiled and run at a speed substantially higher than the previous interpretive systems. Preliminary studies indicate that 300% performance improvement has been reported. This method has additional advantage that it is easier to implement than interpretive systems, because it uses the same compiler and run-time support system during development. Rather than mutating an intermediate form of the program, it must be interpreted. Mutant Schema Generation (MSG) method enables to encode all mutations into one source level program. Then this program is compiled with the same compiler, used during development and is executed in the similar operative environment at compiled-programs speed. While mutation systems based on mutant schemata, need not to provide the runtime semantics and environment. These are considerably less complex and easier to build. A mutant schema has two components, metamutant and metaprocedure set, both are represented by syntactically valid constructs. Metaprocedure is a function that corresponds to an abstract entity in the schema. A statement that has been changed to reflect such a generic form is said to be metamutated. A metamutation is a syntactically legal change that represents other changes. The use of mutant schemata significantly speeds up the mutation analysis. MSG systems are smaller and easier to build, allows to quickly developing mutation tools for a variety of languages. MSG systems also provide more realistic testing. The program tested in the MSG systems retains all or most of its original operational behaviour.

H. *MUTATION CLUSTERING*

As more cluster [11] the mutants, the more lowers the computation cost. By selecting mutants from each cluster, the size of the mutants gets reduced. Further, generate a test set that is mutation adequate for the mutants selected from the clusters. It originates data clustering and mutation analysis, so both mutant set and test set can be reduced dramatically. The clustering algorithms used are k-means clustering algorithm and the agglomerative clustering algorithm. K-means algorithm has number of variations; one of these variations is to select good initial values as the k means clustering depends heavily on the initial selection of the k values. The agglomerative algorithm depends on a threshold value. The data objects are merged pair wise if the distance between them is below the threshold value. All the mutants are represented by the domain of variables to determine the centroid of a cluster. One of the mutants from each cluster is selected randomly for both algorithms and generates the test set for selected mutant by using greedy approach. It has CBT and DDR methods for the generation of test cases automatically. Both of them focus on the fields of variables. CBT takes the algebraic expressions as constraint and DDR takes an initial set for each input. That test set used in killing those mutants. On keeping that test set fixed, generate a new test set of the similar size for other mutants in the cluster. Lastly both the test sets are observed and efficiency of each test set is analyzed based on the mutation score. Those mutants are clustered together, subsequently reducing the size of mutants.

IV. CONCLUSION AND FUTURE WORK

This paper has provided a detailed survey and analysis of trends and results on Mutation Testing. One of the effective techniques for testing is mutation testing. Mutant can be created by changing the syntax of a program. To distinguish the mutant from the original program, an effective test suite is required. In semantic mutation, the language is modified to produce the mutant. There can be misunderstandings in regard to the semantics of the description language. When the syntax of a description is mutated, it is traditional mutation testing. When dealing with language, it is semantic mutation testing. A test case that produces different results when run on the actual program and its mutant is said to be failed. When a test case fails, the mutant is said to be killed. In future mutation testing can be applied to real time applications. Recent developments also include the facility of new open source and industrial tools. These conclusions provide confirmation to support the entitlement that the field of Mutation Testing is now reaching a mature state.

REFERENCES

1. A. Jefferson Offutt, Mary Jean Harrold, Roland H. Untch, “Mutation Analysis Using Mutant Schemata”, ISSE Department George Mason University Fairfax.
2. Axel Hollmann, Christof J. Budnik, W. Eric Wong, Fevzi Belli, Tugkan Tuglular,2016, “Model-based mutation testing—Approach and case studies”, 0167-6423, Elsevier- science and computer programming, doi.org/10.1016/j.scico.2016.01.003.
3. Bradbury J.S., Cordy J.R., and Dingel J., ―ExMAn: 2006, A Generic and Customizable Framework for Experimental Mutation Analysis‖, Proc. Second Workshop Mutation Analysis, pp. 57-62, 2006.

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

4. Chang-ai Sun, Feifei Xue , Huai Liu b , Xiangyu Zhang, 2016, " A path-aware approach to mutant reduction in mutation testing", 0950-5849, Elsevier – Information and Software Technology, .doi.10.1016/j.infsof.2016.02.006.
5. Chevalley P, The´venod-Fosse P, 2002, "A Mutation Analysis Tool for Java Programs", International Journal of Software Tools for Technology Transfer, Vol. 5, No. 1, pp. 90-103, Nov. 2002.
6. Elahe Habibi, Mirian-Hosseinabadi, Seyed-Hassan, 2015, "Event-driven web application testing based on model-based mutation testing", 0950-5849, Elsevier- Information and Software Technology, doi.org/10.1016/j.infsof.2015.07.003.
7. Fan Wu , Jay Nanavati, Jens Krinke, Mark Harman, 2016, Yue Jia , "Memory mutation testing", 0950-5849, Elsevier – Information and Software Technology, doi.10.1016/j.infsof.2016.03.002.
8. 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.
9. Grun B.J., Schuler D., Zeller A., 2009, The Impact of Equivalent Mutants‖, Proc. Fourth International Workshop Mutation Analysis, pp. 192-199, Apr. 2009.
10. Haitao Dan, John A. Clark, Robert M. Hierons, 2011, "Semantic mutation testing", 0167-6423, Elsevier – Information and Software Technology, doi:10.1016/j.scico.2011.03.011.
11. 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.
12. 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.
13. 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.
14. Kwon Y.R., Ma Y.S., Offutt J., 2004, An Experimental Mutation System for Java‖, ACM SIGSOFT Software Eng. Notes, Vol. 29, No. 5, pp. 1-4, 2004.
15. Kwon Y.R., Ma Y.S., Offutt J., 2006, MuJava: A Mutation System for Java,‖ Proc. 28th International Conference on Software Engg., pp. 827-830, 2006.
16. Macario Polo Usaola, Pedro Reales Mateo, May/June 2010, "Mutation Testing Cost Reduction Techniques: A Survey", IEEE Software, vol.27, no, pp. 80-86, doi: 10.1109/ MS.2010.79.
17. Marinos Kintis, Mike Papadakis, Nicos Malevris, 2012, "Isolating First Order Equivalent Mutants via Second Order Mutation", IEEE fifth international conference software testing, 978-0-7695-4670-4/12, doi 10.1109/ICST.2012.160.
18. Mark Harman, Yue Jia, 2009, "Higher Order Mutation Testing", 0950-5849, Elsevier – Information and Software Technology, doi, doi:10.1016/j.infsof.2009.04.016.
19. Moore I., Jester and Pester, 2001, http://jester.sourceforge.net/.
20. Ma Y.S., Kwon Y.R, Offutt J, 2005, "MuJava: An Automated Class Mutation System,‖ Software Testing, Verification, and Reliability", Vol. 15, No. 2, pp. 97-133, 2005.
21. Piattini M., Polo M., Tendero S., 2007, "Integrating Techniques and Tools for Testing Automation‖", Software Testing, Verification, and Reliability, Vol. 17, No. 1, pp. 3-39, 2007.
22. Sang-Woon Kim, Yu-Seung Ma, 2016, "Mutation testing cost reduction by clustering overlapped mutants", 0164-1212, Elsevier- The Journal of Systems and Software, .doi.10.1016/j.jss.2016.01.007.
23. Smith B.H., Williams L., 2007, An Empirical Evaluation of the MuJava Mutation Operators‖, Proc. Third Workshop Mutation Analysis, pp. 193-202, Sept. 2007.
24. Source Forge, ―Jumble,‖ http://jumble.sourceforge.net/, 2007.
25. Tanaka A., 1981, Equivalence testing for Fortran Mutation System Using Data Flow Analysis‖, Georgia Inst. of Technology, 1981.