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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

Computation-based Cost Reduction Techniques and Advanced Platform Support for Mutation Testing

S. Shanmuga Priya

Senior Assistant Professor, Dept. of CSE, New Horizon College of Engineering, Bangalore, Karnataka, India

ABSTRACT: Mutation testing is considered as one of the powerful testing technique that aids in finding bugs in tests and measuring the test adequacy criteria. This adequacy criterion decides the quality of software under test. Some reasons like involving relatively high computation cost during mutant compilation and execution makes mutation testing inefficient. This sets a limitation on its practical impact and not widely preferred in industrial testing. However, these challenges could be resolved. There are two possible ways to reduce the mutant costs. One way is to reduce the number of mutants involved in testing and the other way is reducing the computational cost. The former one is a bit inefficient way because there are wider chances for missing some important mutant as a result of inefficient mutation selection methods. The later one is a good comparatively, as it reduces the computational savings without introducing a need for reducing the mutants. This could greatly increase the efficiency as well as the effectiveness of testing. This paper focuses on the available computation-based cost reduction techniques and advanced platform support for mutation testing.

KEYWORDS: Mutation Testing, Cost Reduction, Strong Mutation Testing, Weak Mutation Testing, Firm Mutation Testing, SIMD, MIMD.

I. INTRODUCTION

Testing is considered as an integral part of software development life cycle process, but still there is an issue in ascertaining the sufficiency or adequacy of test cases considered for testing the System Under Test (SUT). If the tests cannot find a bug in SUT, it cannot be taken for granted that there are no bugs in it. One such technique that is existing to test the tests is mutation testing. Mutation testing is a fault-based testing technique that assesses the efficacy of test cases. It is a white-box based technique and hence it concentrates on the internal structure of the program. The internal structure is deliberately altered by introducing the known faults with a presumption that the tests could uncover it while rerunning a suit of tests against the mutants (versions of altered programs). Figure 1 show the process involved in mutation testing. The entire process can be conceptually divided into five phases:

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

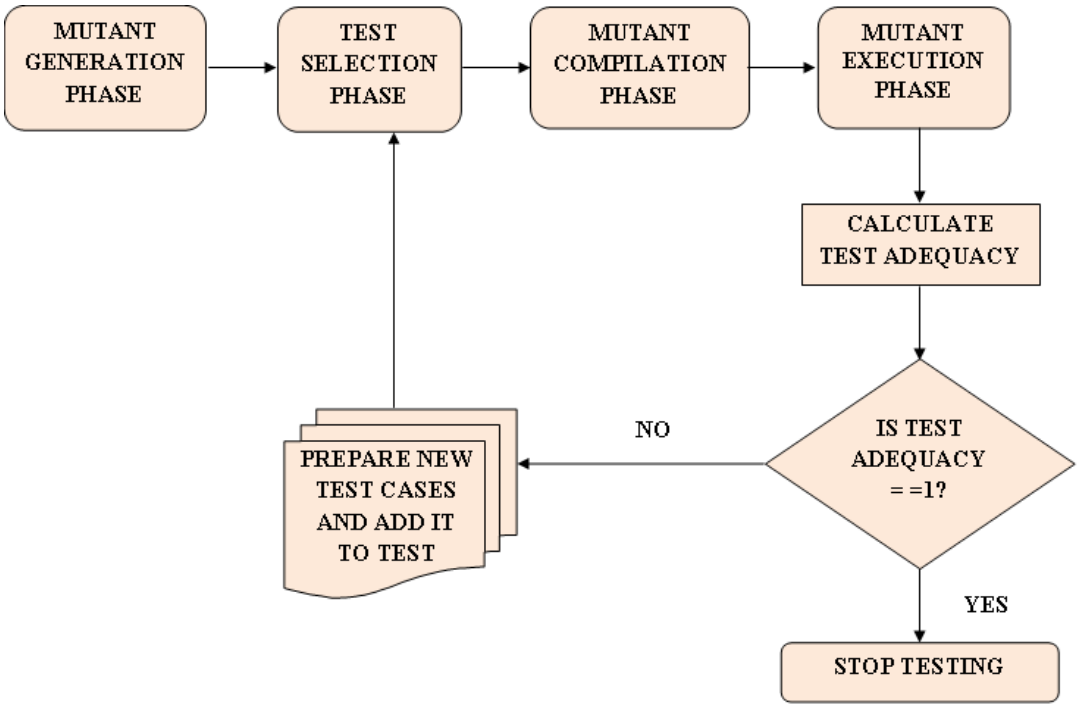

Fig 1. Mutation Testing Process

1. *Mutant Generation Phase* – This phase generates the mutants of an original program that is to be tested. The steps involved in generating mutants are:
   i) Considers the source code, and identify the locations at which the syntactic changes can be made.
   ii) Select equivalent operators for replacing the existing operator in the original source code. This is done in order to introduce faults into the original source code. Such changed version of code is called as mutant.
   iii) Repeat steps i and ii and generate as many mutants as possible for the SUT.
   iv) Mutants can be generated either manually or by using automated tools.
2. *Test Selection Phase* – The tests are selected from the available test suite and generate test data such that it should kill mutants.
3. *Mutant Compiling Phase* – This phase provides mutation compilation solution on the different mutants that are generated from the mutation generation process.
4. *Mutant Execution Phase* – Run mutants against the selected test cases and analyse the result produced test case result with mutant and source code. If the test data kills the mutant, save it as a part of successful test case. This process is repeated for every generated mutant.
5. *Calculating Mutation Score* – This phase calculates the mutation score as follows:
   i) The mutation score is calculated as the ratio of number of mutants introduced to total number of mutants of the program minus total number of equivalent mutants.
   ii) If the mutation score is 1, it indicates that the test case is adequate, else, new tests must be prepared and added to the test suite.
   iii) Repeat phase 3 to 5 is repeated against the tests until the test score becomes 1.

The rest of the sections are organized as follows: Section II gives the various issues in mutation testing, Section III briefs about the reduction by computation cost, Section IV gives the various run-time optimization techniques, Section V throws some insights on advanced platform support for mutation testing, and Section VI gives conclusion.

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

II. ISSUES IN MUTATION TESTING

Even though mutation testing is proving its success ability in evaluating the test set nature, still it has various issues.

A. ISSUES IN MUTANT GENERATION PHASE:

i) Mutants generated must closely resemble the possible error that a programmer might make in the source code. This is a challenging task as the testers must produce meaningful and useful mutants. Testers must pick an appropriate program location and equivalent operator for introducing the syntactic changes. The mutant operators are language dependent. This needs lots of knowledge and practice.

ii) Almost all executable statement in the original program can be mutated with several possible mutation operators. One major issue that is always prevalent in mutation testing is huge number of mutants generated in generation phase even for small code under test. Few such evident are: Offutt et al. [1] considered some 10 programs from 10 to 48 executable sentences. This produced around 183 to 3010 possible mutants. Mresa and Bottacia [2] considered some 11 java programs for their work with having 7 lines of codes produced around 3211 mutants. Macario Polo and Mario Piattini [3] also experienced in producing huge number of mutants for small code. Myers [4] took the triangle type problem which had 61 lines of code, and it produced around 262 mutants.

iii) Generating all possible mutants is a time consuming process if all these mutants must be generated manually. Automated mutants production might give a solution for producing mutants.

B. ISSUES IN TEST CASE SELECTION PHASE:

The decisions taken at this phase is very much essential, as it will directly reflect in deciding the performance of mutation execution phase. An apt set of test cases need to be deduced such that it can kill all mutants. Test selection is left to the testers' choice either it can be selected manually or can be automated. However, this can largely be automated [5], [6], [7].

Generally, when picking test cases for execution, it's better to pick relevant tests for each mutant. As this may lead to an in efficient testing, mostly a large portion of the test suite or the entire test suite is selected and run against each mutant. One implementation that can improvise this phase is to arrange the test cases in an optimal running order which might reduce the running cost.

C. ISSUES IN MUTANT COMPILING PHASE:

Generating huge number of mutants for even simple codes involves significant cost in compiling all generated mutants, but this is inevitable.

D. ISSUES IN MUTATION EXECUTION PHASE:

As it's necessary to execute the tests against each mutants and original program, huge number of test cases must be executed which consumes time. For example, consider a simple application that consists of a class with 100 mutants and 350 test cases, might require around 35,000 executions (100 * 350 = 35,000). This may take so much of database access, etc. Also, it inquires huge time to compare the output produced. If the mutants are killed, it can be removed from further comparisons. Alive mutants and equivalent mutants must be found properly.

New tests should be prepared such that it must kill those two mutants and it should be added to the test suite in order to enhance its adequacy. Newly designed test cases must be meaningful and it must have potential to kill the equivalent and alive mutants. It should also take care of avoiding redundant test cases while introducing new tests.

III. REDUCTION BY COMPUTATIONAL TECHNIQUES

Even though mutant reduction technique is good in certain aspects for reducing the costs, it has its own disadvantage. Some compromise must be made with mutant selection approach. Inefficient mutant selection algorithms might force to discard effective mutants, which in turn affects the quality of test and makes mutation testing meaningless. So reduction by computational cost is considered as an alternative solution to make mutation testing effective. This section gives a glimpse on the three techniques used for optimizing mutant compilation and execution process. They are strong mutation testing, weak mutation testing and firm mutation testing.

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

A. STRONG MUTATION TESTING

DeMillo et al. [8] originally formulated strong mutation testing aka traditional mutation testing. For a given original program P, a mutant of the program R, is said to be killed if and only if the mutant R gives a different output from the original program P considered for testing. Strong mutation testing continues mutant execution until it completes the entire program and compares the subsequent outputs. It is considered as waste of time to execute the entire program under test. Strong mutation testing is suitable for unit-testing technique, but it endures from expensive computational cost. However, the execution of the strong mutation can be optimized. Not restricted to the traditional programming language testing, strong mutation testing finds its place in testing the web service based languages. Antonia et al. [9] and Estero et al. [10] have also given their contribution for strong mutation testing. They have given wider varieties of WS-BPEL operators.

B. WEAK MUTATION TESTING

Howden [11] was the first to propose weak mutation testing that could actually reduce the computation cost in terms of time and space. The underlying concept of weak mutation testing considers two conditions: *reachability*, whether a faulty code can be reached and executed, infection *point*, execution of the faulty code leading to an incorrect internal state of the program.

In this testing, the states of both original program P and the mutant program R are compared at a predetermined point of execution, which is immediately after the mutated statement execution. If the states are different, the mutant is killed, otherwise alive. The results are recorded and execution can be terminated. It is not required to execute the entire program. The major advantage of weak mutation is that it reduces the amount of execution for distinguishing each mutant. The major disadvantage is there is no guarantee that this method can detect complicated defects, introduced by certain mutants. More over at it does not necessarily follow the path all the way to its termination; there is a possibility, that this path may mask certain errors.

Pawan Kumar Chaurasia [12] has given a review on mutation testing. An approach for weak mutation testing can also be found in [13] and [14]. Weak Mutation testing plays a major role in uncovering the faults in web based languages as well. Panya Boonyakulsrirung and Taratip Suwannasart [15] proposed a weak mutation testing tool for Web Service-Business Process Execution Language (WS-BPEL) that supports generating mutants, executing the test cases against the created mutants, also specifies the status of the mutants whether it's alive, killed or equivalent mutants. Boonyakulsrirung et al. [16] proposed a weak mutation testing framework for WS-BPEL. The authors analysed the different categories of mutation operators proposed in [9] and [10].

C. FIRM MUTATION TESTING

Strong mutation testing is effective in producing results, but it involves huge computational cost. Weak mutation testing drastically reduces the number of test cases generated, but it is considered cost effective only to the components of the programs. Woodward and Halewood [17] was the first to propose firm mutation which is a combined approach that overcomes the disadvantages of strong and weak mutations. Firm mutation testing is more flexible technique and is computation less expensive one. It allows users-selectable portion of code to be tested, also mutant operators can be selected according to the code portion, and results can be compared after execution. This user-selectable flexibility is considered as the biggest advantage of firm mutation testing as they are given with a choice to determine the criteria needed to kill a mutant.

IV. RUN-TIME OPTIMIZATION TECHNIQUES

Run-time optimization technique can be performed either interpreter-based or compiler-based. There are some six run-time based optimization techniques available in the literature. This section gives a view on the various such optimization techniques.

A. INTERPRETER-BASED TECHNIQUE

This technique was first proposed by Offutt [18] in 1987 for reducing the execution cost of mutants. Figure 2 shows the underlying idea of interpreter-based technique. The basic procedure followed is to translate the original program into an intermediate code, and the mutants were generated from that intermediate form. The interpretation cost was taken into account. The main advantage of this technique is it's efficient and flexible when program under test is considerably small. Because of the well know nature of interpretation, the consequence is, this technique is very slow.

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

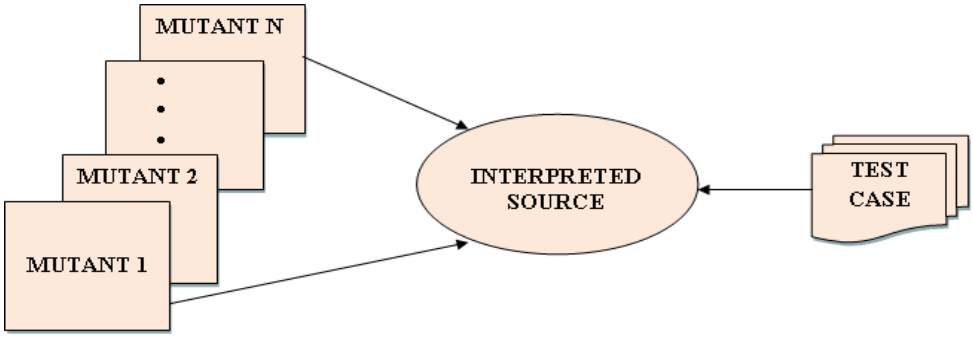

Fig 2. Interpreter-based technique

B. COMPILER-BASED TECHNIQUE

Figure 3 shows the compiler-based technique. In this technique, the generated mutant code is compiled first and is converted into an executable file. Then such compiled mutant will be executed with the test cases. Execution of compiled binary code is much faster than that of the interpretation. The running cost and sum of compilation cost gives the overall cost of this technique. This technique can be found in the work of Delamor [19], [20]. Due to high compilation, larger programs under test may have its own speed limitation [21].

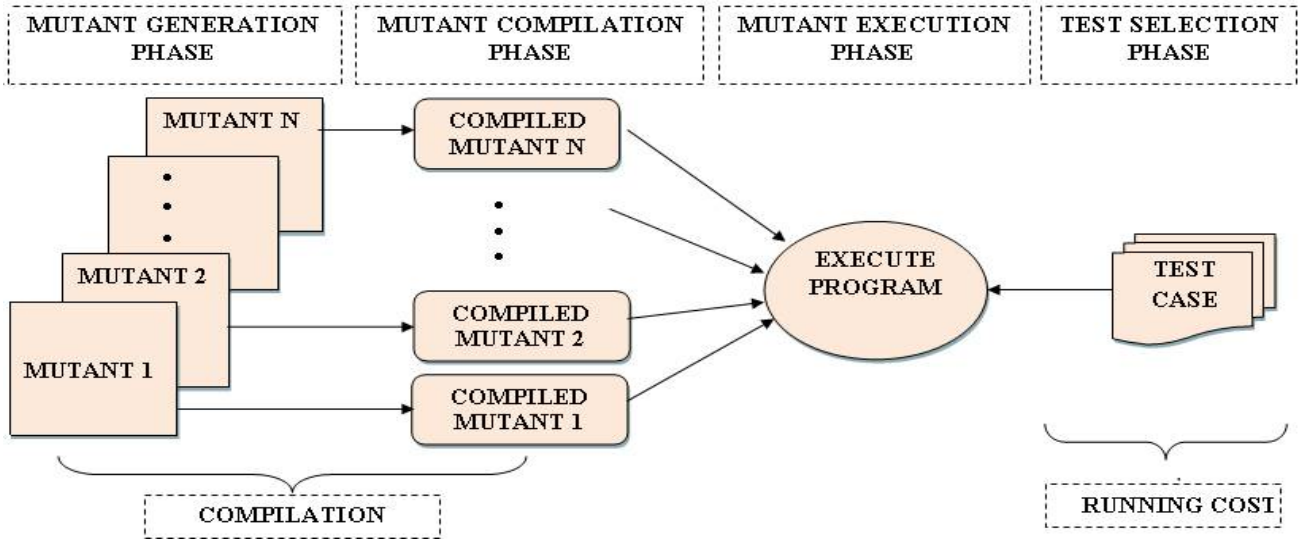

Fig 3. Compiler-based technique

C. COMPILER-INTEGRATED TECHNIQUE

The difference between original program and mutant is a minor syntactic change. Compiling each created mutants separately inquires much cost which can be considerably reduced by using a compiler-integrated technique proposed by DeMillo et al. [22], [23]. An instrumented compiler was designed in compiler-integrated technique to compile the mutants; it significantly optimized the compilation cost of the traditional compiler-based technique.

D. MUTANT SCHEMA GENERATION

In order to reduce the cost overhead in traditional interpreter-based techniques, mutant schema generation approach can be implemented. Compiling each mutant separately involves huge cost in terms of time and space. Instead of separately compiling all such created mutants, a metaprogram can be generated. This metaprogram can be used to represent all possible mutants. It is sufficient to compile the metaprogram once and run against the test set

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

instead of compiling and executing all possible mutants. This significantly reduces the compilation cost and overall runtime cost. It is also a speedier technique when compared with the traditional interpreter-based one. [24].

Ronald et al. [25], [26] designed a Mutant Schema Generation method that has encoded all mutations into a single source level program, compiled and executed under the same developed environment. It was identified that their system was able to operate at the source-level, and more portable (can easily be moved from machine to machine, or compiler to compiler).

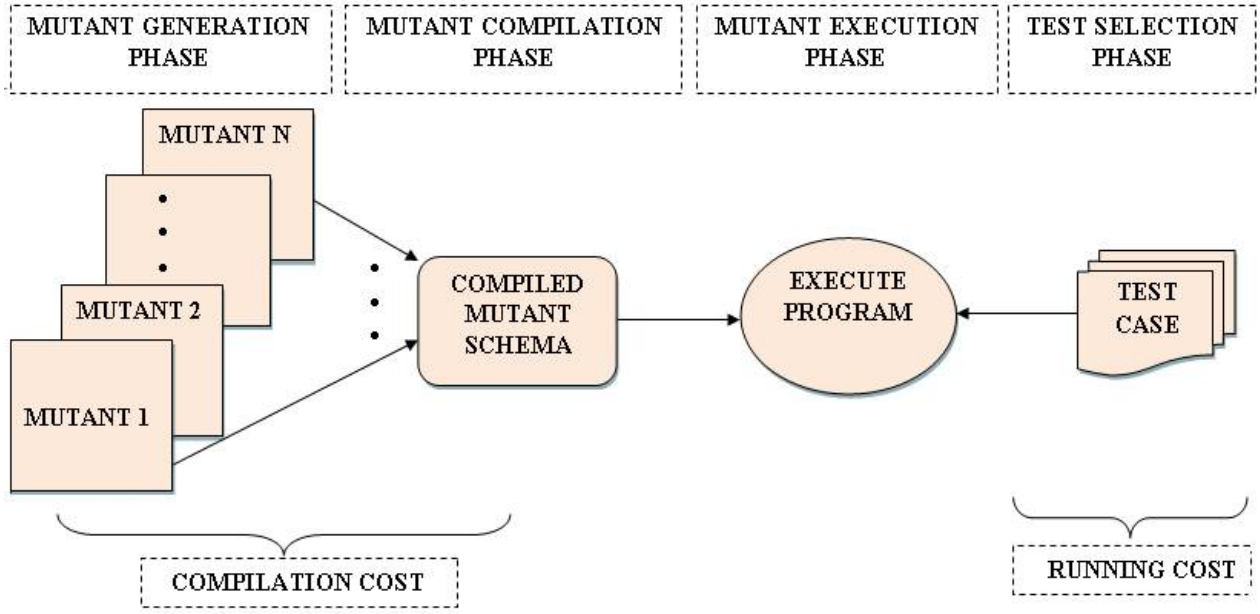

Fig 4. Mutant schema generation technique

E. BYTECODE TRANSLATION TECHNIQUE

Another way to reduce the compilation cost is to employ Bytecode translation technique. This was first proposed by Ma et al. [27], [28]. Instead of producing mutants from the source code an alternative approach was used. The mutants were generated from the compiled object code of the original program subjected to testing. Involving such method eliminates compilation needs and the bytecode mutants can be directly executed, thus saving the compilation cost.

F. ASPECT-ORIENTED MUTATION TECHNIQUE

Bogacki and Walter [29], [30] in 2006 proposed this aspect-oriented mutation approach. This approach eliminates the necessity of compiling each mutant separately. Instead, an aspect patch was generated and it was run twice, once with original program and then with mutants.

V. ADVANCED PLATFORM SUPPORT FOR MUTATION TESTING

Mutation testing is not only restricted to optimize by compiler and interpreter techniques, it can be applied to architecture-level and the overall computation cost involved among many processors can also be reduced. Several works has been made for implementing mutation testing on high-platform computer systems. Parallel mutation testing works has been suggested for vector-based processors Single-Instruction-Multiple-Data (SIMD) stream and Multiple-Instruction-Multiple-Data (MIMD) machines.

Mathur and Krauser [31] proposed mutation testing on vector based processor system. Multiple generated mutants can be executed in parallel so that execution time could be reduced. Krauser [32] implemented the cost

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

reduction approach under SIMD machines. The authors implemented a mutant unification approach, where the mutants of same type were grouped together and that groups were handled by different processors in the SIMD system, thus executed in parallel. An algorithm to improve the mutation testing in parallel machine can be found in Fleyshgakker [33] and Weiss [34]. Offutt [35] and Choi and Mathur [21] have proposed a technique for optimizing mutation testing cost in MIMD machines. Zapf [36] proposed a novel idea, where the mutants could be executed independently in a network based environment.

Researchers are working on introducing the virtualization concept for executing mutants in order to increase the speed of execution. Durelli et al. [37] used virtualization-based implementation for mutation testing and achieved speedups of 89% in some cases.

Even though these advanced techniques are good enough in solving the computation cost involved in mutation testing, they have their own demerits. They are not cost effective particularly, for those companies on limited budget, as it require specialist equipment.

VI. CONCLUSION

One major issue that evades making mutation testing as a viable testing method is its high computational cost in executing huge number of mutant against a test set. Another possible issue is due to the equivalent mutant issue. By crossing all these issues, the mutation testing is still proving to be a good testing technique for identifying the hidden faults which sometimes a test case fails to reveal. A great deal of past work is available in the history reveals that still there is a hope to reduce the computational expense charged by the mutants. Hope, these techniques could improve and help the chances of mutation testing getting adopted in industrial testing techniques.

REFERENCES

1. Ma, Y.S., Offutt, A.J., Kwon, Y.R, "MuJava: A Mutation System for Java", International Proceedings of 28th International Conference on Software Engineering, pp. 827-830, 2006.
2. Mresa, E.S., Bottaci, L., "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study", Software Testing, Verification and Reliability, pp. 205–232, 1999.
3. Macario Polo and Mario Piattini, "Mutation Testing: Practical Aspects and Cost Analysis", 2011.
4. Myers G. J., "The Art of Software Testing", 1979.
5. Offutt J., "Automatic Test Data Generation", 1988.
6. Ayari, K., Bouktif, S., Antoniol, G., "Automatic Mutation Test Input Data Generation via Ant Colony", In Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM, London, England, pp. 1074–1081, 2007.
7. Blanco, R., Tuya, J., Adenso-Díaz, B., "Automated Test Data Generation using a Scatter Search Approach", Information and Software Technology 51, 708 –720, 2009.
8. Richard A. DeMillo, Richard J. Lipton and Frederick Gerald Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer Computer", 11(4), April 1978.
9. Antonia Estero-Botaro, Francisco Palomo-Lozano, and Inmaculada Medina-Bulo, "Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions", Department of Computer Languages and Systems, University of Cadiz, Spain.
10. A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Mutation operators for WS-BPEL 2.0,", in ICSSEA 2008: 21st International Conference on Software & Systems Engineering and their Applications, Paris, France, 2008.
11. William E. Howden, "Weak Mutation Testing and Completeness of Test Sets", IEEE Transactions on Software Engineering, 8(4), July 1982.
12. Pawan Kumar Chaurasia, "Mutation Testing: A Review", Journal of Global Research in Computer Science, Volume 5, No. 2, pp. 33-36, February 2014.
13. Natthapol Thaisakonpun and Taratip Suwannasart, "Mutation Testing for Expression Modification Operator of BPEL" Software Engineering Laboratory, Centre of Excellence in Software Engineering, Faculty of Engineering, Chulaongkorn University, Bangkok, Thailand.
14. A. Jefferson Offut and Stephen D. Lee, "An Empirical Evaluation of Weak Mutation", Department of Information of Informal and Software Systems Engineering, George Mason University Fairfax, 1996.
15. Panya Boonyakulsrirung and Taratip Suwannasart "WeMuTe – A Weak Mutation Testing Tool for WS-BPEL", Proceedings of the International MultiConference of Engineers and Computer Scientists, Hong Kong, Vol. 1, 2012.
16. Panya Boonyakulsrirung and Taratip Suwannasart, "A Weak Mutation Testing Framework for WS-BPEL", Computer Science and Software Engineering (JCSSE), Eighth International Joint Conference, pp. 313-318, 2011.
17. Martin R. Woodward and K. Halewood, "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues", Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88), July 1988.
18. Jefferson Offutt .A and King K. N., "A FORTRAN 77 Interpreter for Mutation Analysis", ACM SIGPLAN Notices, 22(7), 1987.
19. Marico Eduardo Delamoro, "Proteum – A Mutation Analysis Based Testing Environment", University of Sao Paulo, Brazil, 1993.
20. Marico Eduardo Delamaro and Jose Carlos Maldonado, "Proteum – A Tool for the Assessment of Test Adequacy for C Programs", Proceedings of the Conference on Performability in Computer Systems (PCS'96), New Jersey, 1996.
21. Byoungiu Choi and Aditya P. Mathur, "High-Performance Mutation Testing", Journal of Systems and Software, 1993.

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

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

22. Richard A. DeMillo, E. W. Krauser and Aditya P. Mathur, "Compiler-Integrated Program Mutation", Proceedings of the 5th Annual Computer Software and Applications Conference (COMPSAC'91), Japan 1991.
23. Krauser, E.W, "Compiler-Integrated Software Testing", Ph.D Thesis, Purdue University, 1991.
24. Ronald H. Untch, "Mutation-based Software Testing using Program Schemata", Proceedings of the 30th Annual South East Regional Conference (ACM-SE'92), North Carolina, 1992.
25. Ronald H. Untch, A. Jefferson Offutt and Mary Jean Harrold, "Mutation Analysis using Mutant Schemata", Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93), 1993.
26. Ronald H. Untch, "Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method", South Carolina, 1995.
27. Jefferson Offutt .A, Yu-Seung Ma and Yong-Rae Kwon, "An Experimental Mutation System for Java", ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, pp. 1-4, 2004.
28. Yu-Seung Ma, A. Jefferson Offutt and Yong-Rae Kwon, "MuJava: An Automated Class Mutation System", Software Testing, Verification and Reliability, Vol. 15, No. 2, pp. 97-133, 2005.
29. Bogacki, B., Walter, B., "Evaluation of Test Code Quality with Aspect-Oriented Mutations", International Proceedings of Seventh International Conference on eXtreme Programming and Agile Processes in Software Engineering, pp. 202-204, 2006.
30. Bogacki, B., Walter, B., "Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing", Software Engineering Techniques: Design for Quality, Springer, 2007.
31. Aditya P. Mathur and Edward William Krauser, "Mutant Unification for Improved Vectorization", Purdue University SERC-TR-14-P, 1988.
32. Edward W. Krauser, Aditya P. Mathur and Vernon J. Rego, "High Performance Software Testing on SIMD Machines", IEEE Transactions on Software Engineering, 1991.
33. Vladimir N. Fleyshgakker and Stewart N. Weoss, "Efficient Mutation Analysis: A New Approach", Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94), 1994.
34. Stewart N. Weiss and Vladimir N. Fleyshgakker, "Improved Serial Algorithms for Mutation Analysis", ACM SIGSOFT Software Engineering Notes, 1993.
35. A. Jefferson Offutt, Roy P. Pargas, Scott V. Fichter and Prashant K. Khambekar, "Mutation Testing of Software using a MIMD Computer", Proceedings of the International Conference on Parallel Processing, 1992.
36. C. N. Zapf, "A Distributed Interpreter for the Mothra Mutation Testing System", Clemson University, 1993.
37. Durelli, V.H.S., Offutt, J., Delamaro, M.E., "Toward Harnessing High-level Language Virtual Machines for Further Speeding up Weak Mutation Testing", IEEE Bibliography Fifth International Conference on Software Testing, Verification and Validation, Canada, pp. 682 – 690, 2012.