A Survey on Different Approaches for Efficient Mutation Testing

MeghaJhamb*, AbhishekSinghal*, AbhayBansal*

* Department of CSE, ASET, Amity University, Uttar Pradesh, Noida

Abstract- Before delivering the software to the customer the testing of the software is an essential task. Many testing have been used by the software testers in order to check the functionality and quality of software, one of these testing is Mutation testing. Mutation testing is the testing that has received a large amount of investment.In this paper we have reviewed various proposed methods that are capable of performing Mutation testing of software from the consumer's perspective and able to manage the quality requirement in the application.

Index Terms- MutationTesting, Direct Graph, Regular Grammar, K-sequence Coverage, Mujava;

I. INTRODUCTION

In the recent years the use of computer aided software has become a common task. The people are becoming more dependent on the computer based software and computer application. In order to full fill the needs of people the software development companies are designing the various small sizes to big size and complex soft ware's. Since the use and demand of software is increasing day by day, this is the responsibility of the software development team to deliver efficient and right and testified software to their customer. Thus to achieve this testing of software is an important task. The testing of software is defined as the process of checking software that whether it is performing the function for which it has been designed, and as well as is it meeting the requirements of the customers, without testing the software cannot be delivered to the system. In short testing is a process of delivering error free software to the customer.

Now there are many software testing researchers who have proposed a number of different types of testing like white, black box and model testing. One of the proposed and most researched testing is Mutation Testing [2].

There are some terms in the testing that need to be defined before discussing any other thing about testing or mutation testing. These terms are test suite and coverage [10].During the process of testing a number of test cases are designed to test the software, the collection of these test cases is known as Test Suite.The part of whole code of the software that a test suite can test is known as coverage.When the code become large and the results of test are not good then selecting the simple testing technique becomes invalid. In such case the tester cannot decide that whether there is any problem in code or the test suite selected is wrong. In such situation mutation testing can be used because mutation testing has been designed to overcome this problem only.

Mutation testing is a testing in which the original code is changed and the changed code is testified to check whether there is a problem in code or in test suites. This changed code is known as MUTANT. When mutant is tested under the selected test suite if the test suites are right then they will catch the error in the Mutant, it means that there is an error in the code and the test suites selected are fine but if the test suites cannot find the error in the mutant then it means that the selected test suites has some wrong test cases and those test cases need to be changed. When one of the test cases identifies the error in the mutant then it is said that mutant has been killed. This is the whole procedure of the Mutation Testing.

We can perform the mutation testing using various methods, the most common and easy way of performing mutation testing is the use of Directed Graph. In directed graph the whole program code is converted into the directed graph having a number of nodes and arcs but the Directed graph has some its own drawbacks during its use in the mutation testing.

Thus, in this paper we have discussed about the various proposed such techniques for performing the Mutation Testing and one of the tool used for performing mutation testing in various Object oriented programming language.

Rest of the paper is organized as: description of Directed Graph Model in section II, Regular Grammar Model as in section IIIand Semantic Mutation Testing is explained under section IV, Mutation testing in Java is summarized in V and MuJava tool in the section VI.Finally the future work and a conclusion of the discussion are provided in last section

II. DIRECTED GRAPH

It's the foremost preliminary step for proceeding with any research work writing. While doing this go through a complete thought process of your Journal subject and research for its viability by following means:

A directed graph (DG) has V set of finite nodes is a where tuple D = (V, A, S, F) , A has ordered pairs of elements of V which is represented as a finite set of directed graphs , S is a set of start nodes and F is a set of finish nodes. Both S and F belongs to V [3].

This is the definition of directed graph when it is used with the mutation testing. When a program code is converted into a directed graph the one statement of code is represented by the one node of the graph and the paths following that statement is shown by the arcs of the graph. While constructing a directed graph some of the nodes in the graph should be distinguished as start and finish node to clear the usefulness of DG and to show the start and end of the code. Therefore S and F are the Start and

Finish node in the Directed Graph. There are some definitions in context to directed graph in mutation testing.

In a directed graph a node v belonging to V is said to be useful only when there is a path from a start to the finish node [3].

For a given directed graph, a node v belonging to V, related to v strictly preceding nodes are those nodes which occurs in all the paths from these nodes to the finish nodes [3].

For a given directed graph, a node v belonging to V, related to v strictly succeeding nodes are those nodes that occur only in the paths from v to the finish nodes [3].

For implantation- oriented, white box testing nodes of the DG to be covered generally is represented by the statements of SUT (Software under Test) and with arcs sequence of statements is explained [5].
For specification, in the black box testing, behavioral events of SUT are represented by the nodes of DG and arcs represent the sequences of those events [6]. This is how the directed graph is constructed for a given code.

But in mutation testing negative testing is also performed it means it is tested that the software is not doing anything it is not supposed to do. For this [3] have proposed specific manipulation operators for the graphs that models the SUT. These operators are discussed as below:
The directed graph consists of nodes and arcs so it is obvious that the manipulation can be done either in nodes or the arcs. Therefore there are two types of the elementary manipulation operator that can be applied on arcs and nodes. These are insertion and omission. These are defined as below:

a) Node Insertion –

Here a new node v is added to the DG together by the operator with possibly nonzero numbers of arcs, and connecting this node with the remaining nodes. Both the sets A and V are updated after the node insertion.

b) Node Omission-

The existing node v is deleted by the operator from the DG along with arcs, ingoing and outgoing from the deleted nodes. After node omission both the sets A and V are updated.

c) Arc Insertion-

This operator inserts or adds a new arc to the DG, after this a new set A of arcs is formed.

d) Arc Omission-

This operator deletes an existing arc ‘a’ from the DG and the set of arcs A is updated. It may result in some nodes with no ingoing and outgoing arcs.

This is how manipulation is performed in the directed graph.
In every testing task there is a main focus on the coverage. So in context of directed graph coverage is defined as three practical coverage criteria and these criteria are defined as below:

1) Node Coverage:

Given a graph DG and a set of strings B, is said to cover a node v, if v occurs at least in one of the strings in B. If the set of string B covers all nodes in V, then it is said to achieve node coverage.

2) Edge Coverage:

Given a graph DG, a set of strings B is said to cover an edge (u, v) of A, if the sequence uv occurs at least in one of the string in B. If the set of string B covers all edges in A, then it is said to achieve edge coverage.

3) Path Coverage:

Given a graph Dg and a set of string B, is said to cover a path of length k if sequence u1u2---- uk occurs at least in one of the string sin B.

While performing manipulation in the model here it is directed graph one should keep in mind that manipulation operators should not invalidate the model and should take necessary required steps to convert invalid model into valid model. For directed graph model while constructing directed graph model of a SUT, considering the system semantics we should determine start and finish nodes. For doing this [3] have proposed two main approaches:

Fix the set of start nodes and finish nodes and do not allow any sequence of manipulation operations which violate the usefulness of any node in DG.

After performing some manipulation operations, if the resulting DG is invalid, then to validate usefulness of all the nodes select a new and different start and finish node and transform it to the valid one.

This is all about the Direct Graph model for performing mutation testing.

III. REGULAR GRAMMAR

In this section we will discuss that how regular grammar can be used for performing mutation testing and how a directed graph can be converted into a Regular Grammar. So in this section we are discussing the regular Grammar Model of performing Mutation Testing.

The Directed Graphs can only be used to model regular systems. The test generation algorithms based on DGs can be viewed as still being in their starting point means the existing few are relatively slow and they are memory consuming. To overcome such issues of Directed Graphs, regular grammar can be used to model the application, so here is an introduction of a new approach for modeling the mutation testing applications.

The basic definition of the grammar is a tuple (N, E, P, S) where, N is a finite set of non terminal symbols, E is a finite set of terminal symbols, P is a finite set of production rules and the S belongs to N and is a distinguished nonterminal start symbols.

To construct the regular grammar model for mutation testing the Directed graph is converted to the Regular grammar. Since the FSA, DGs and RGs equivalently describe regular grammar, it is possible to construct regular grammars from the directed graphs. In the constructed regular grammar from a directed graph the production rule of grammar represents the arc of the graph and terminal and non terminal symbols represent the nodes of the graph. According to [3] the constructed grammar has the following properties that make it more efficient model than directed graph

- G is right RG (so it is unambiguous).
- nt(x) = Rx is a bijection to N\{S} via E.
- nonterminal S appears only on the left side of the production rule.
- Each node in V corresponds directly a terminal in E.
- Each arc in V corresponds to a production rule in P (including the pseudo arcs used to mark start and finish nodes).
- All terminal symbols in the grammar are useful if and only if all nodes in the DG are useful.

Similar to the directed graph, the regular grammar also have some special definitions and the manipulation operators. These definitions are discussed as below:

- For a given grammar (N, E, P, S), a terminal symbol r and a non terminal symbol R. A terminal symbol r is said to be useful if it occurs in at least one string in L(G) and the non terminal symbol R is said to be useful if a rule of the form (N union E) * R(N union E) * $\longrightarrow$ …. Is used in a derivation S $\longrightarrow^*$ … of at least one string in L (G).

- For a given grammar (N, E, P, S) and a terminal symbol r, then the strictly preceding terminal symbols for r are all those terminal symbols such that r occurs in all the strings in L (G) in which these terminal symbols also occur and r occurs after these symbols.

For a given grammar (N, E, P, S) and a terminal symbol r, strictly succeeding terminal symbols for terminal symbol r are those terminal symbols which only occur in the either one or all the strings where r also occurs and they occur after r.

Above we have discussed the special definitions of the regular Grammar, now further we will discuss the manipulation operators of the Regular Grammar Model.

a) Arc Insertion-

The arc insertion operator adds a new arc in the directed graph and in the grammar arc is represented by the production rule. Therefore when an arc is inserted in the DG then RG should be updated with a new production rule to show the insertion of this arc.

b) Arc Omission-

The arc omission operator deletes an existing arc from the directed graph and in the regular grammar an arc is represented by the production rule. Therefore when a arc is deleted from DG then a production rule must be removed from the RG. This operation may violate the usefulness of some terminal symbols so some extra care is needed while performing this operation.

Node Insertion-

The node insertion operator adds a new node in the Directed Graph and when a new node is added in the DG then new arc ingoing and outgoing form that node are also inserted in the DG. In Regular Grammar a node of DG is represented by the terminal and non terminal symbols, thus when a node is inserted in DG a new non terminal symbol and a new non terminal symbol is added to the RG. And due to the addition of terminal symbols and non terminal symbols in the RG the production rules are also added in the RG to show the arcs ingoing and outgoing from the nodes.

c) Node Omission-

The operation of arc omission deletes a nodes from the graph due to which the arc ingoing to and outgoing from this nodes are also deleted. Therefore the node omission operation is based on arc deletion of arcs from the graph.

Thus when deleting a arc from DG, a production rule is deleted from the RG and proper steps should be taken to first the arc is deleted to perform removal of the production rules, and then the isolated terminal symbols and non terminal symbols are removes from the grammar.

Now we will discuss the coverage criteria included in the Regular Grammar. The coverage criteria are discussed as below:

I. Terminal Symbol Coverage:

For a given grammar G and a set of sting A is said to cover a terminal symbol e, if e is present in at least in one of the string in A. if the string A contains all the terminal symbols of E then it is known as Terminal Symbol Coverage.

II. Production Rule Coverage:

For a given grammar and a set of strings A, then A is said to cover a production rule p, if p is used at least in one of the derivation of A. if the set of string A contains all the production rules belonging to P (set of production rules) then it is known as Production Rule Coverage.

This is all about the Regular Grammar model for Mutation Testing.

IV. SEMANTIC MUTATION TESTING

In this section we are going to introduce a new approach behind the mutation testing. The simple mutation testing about which we have given a small description works on the syntax of the program or code of the software, means it focus on the code not the semantic of the code it only check the syntax not the semantic of the language used. In such condition the number of mutants gets increased in number which becomes difficult to execute practically. There is a need of a change in the standard mutation testing [4]. For this they have proposed a new mutation testing that is Semantic Mutation Testing. This testing focus on the

semantic of the language used means it performs the operations of mutation on the semantic of the language used. In this section we are going to describe in brief why we need semantic mutation testing over standard mutation testing and what the basic concept behind the Semantic Mutation Testing is.

A. Need of Semantic Mutation Testing

When software is developed then it passes through various phases of software development Life cycle model, in each step of life cycle model the team members give the different description to the software and uses different languages in this situation the language vary from step to step. When the transformation of prototype model of software to target model of software take place the description again changes and there come different descriptions of software from this different misunderstanding regarding the actual description of software arises, in such case our traditional mutation testing cannot work efficiently in detecting the error in code and misunderstanding regarding the description of the software. To overcome this problem of Mutation Testing, a modified version of mutation testing "Semantic Mutation Testing" was proposed by [4]. This works on the semantic of the language used for the program. The semantic is modified to generate the mutant means mutant operations are applied on semantic rather than description. The working and detailed concept of Semantic Mutation Testing is described in the next part of this section.

B. Concept Behind Semantic Mutation Testing

In Standard Mutation Testing the mutation operators are applied at syntactic level in does not focus on the semantic misunderstanding generated by semantic mistakes as explained above. For this semantic mutation testing was introduced. In this testing mutation operations are applied on semantic and it explores the variance in the semantics.

The code which is under test is represented by a description. Take an example, a description D is written in language with semantic S, the behavior of the description is defined by an ordered pair (D, S). The standard mutation testing will apply the mutation operations on the D that is the description under test. Thus the mutant for (D, S) can be represented as (D, S) $\rightarrow$(D', S) where D' is the change in the D.
On the other hand the Semantic Mutation Testing will apply the mutant operator in the semantic of the language that is S. thus the mutant obtained for the (D, S) will be represented as
(D, S) $\rightarrow$(D, S') where S' is the semantic obtained after applying mutation operations.
Now how the Semantic Mutation Testing kills a mutant. For a given D of (D, S) after applying mutation operations on S there are two interpretation one its explanation under S and second its explanation under S' and we will call these as Ds, Ds' respectively. For a test case say t, Ds (t) will denote the behavior produced when applying t to D under semantics S and Ds' (t) will denote the behavior produced when applying t to D under semantics S'. Then mutant (D, S') is killed by test case t only when Ds (t) = Ds' (t).

Further, if for all t we have Ds (t) = Ds' (t) then this mutant (D,S') is said to be an equivalent mutant.
From here it is clear that, the notions of behavior, equality of behavior, and thus the language used by the testers will decide the killing of the mutant. Along with this, the semantic mutation made and the description under test are the factors that will decide the property of killing the semantic mutant [4].

V. MUTATION TESTING IN JAVA

In this section we are going to discuss a how mutation testing is different for the code written in language java. In the basic mutation system there are two main parts. In the first section the original code is converted into the mutants or also known as mutation programs, this is done by using the mutation engine that has a number of mutation operators. In the second section the test cases are generated and are run as input for the respective program [7]. Hence the accuracy of the test cases is decided if the test cases detect the fault in the mutant.
The major deficiency of the traditional mutation testing tool is that they lack in automation for major parts of the process [11]. The important tasks like entering test cases and then running the test case on original input and then checking the results, and after that running the test case on mutants and checking the results are very human-intensive [7]. In order to remove this lacking of tool, schema generator was introduced and the MuJava tool is based on this advancement. The concept behind schema generation is production of meta-mutant. It incorporates all the mutants on an original code in a single code/program.
In the mutation testing for code in object oriented language like java the mutation system has two types of mutation operators
a) Traditional mutation operators
b) Class level mutation operators

The traditional mutation operators are those which are generated from procedural language but the class level mutation operators' works on the features of OOP like inheritance, polymorphism, Data binding, overloading.
The table 1 shows some of the traditional as well as class level mutation operators [13].

TRADITIONAL OPERATORS	CLASS LEVEL MUTATION OPERATORS
ABS	AMC
AOR	PNC
LCR	OAN
ROR	JTD (java specific)
UOI	EOA

Table1. Different Mutation operators

For JAVA 24 class mutation operator were found by Ma, Kwon and Offutt for testing object oriented and integration issues [7]. The class mutation operators are applicable at different levels as in intra-method, inter-method, interclass and intra-class. In reference [7] author has proposed a mutation testing engine for

the JAVA. In the next section we are going to discuss the MuJava tool for performing mutation testing in JAVA.

VI. MUJAVA TOOL

In this section we will discuss a tool "MuJava" which support the object oriented mutation testing for Java programs [7]. This system was generated by the Yu Seung Ma and Jeff Offutt initially it was known as JMutation, a mutation system for java programs but later it was changed to MuJava. The main concept of schema generator on which the MuJava is based is Schema Generation [14], in this the Meta mutant is generated which incorporates all the mutants on an original code in the one program. The MuJava does not generate test cases, it only generate the mutants and run the test cases supplied by testers [8].

The MuJava tool consists of three phases:

1) Mutant Generation:
In this phase the mutant operators are applied on the original piece of code to generate the mutants. In the tool interface there is an option of "Generate" button where tester can select the file for which they want to create mutants and can choose which mutant operator is to be applied on file.

2) Analyze Mutant:
n this phase mutants are analyzed. In MuJava tool there is an option "Mutant Viewer" to view the mutant created. There is an option where the user can analyze the generated mutant against the original code. When the class level mutant operators are applied MuJava generates mutants in two different ways (MSG and byte code translation)

3) Run Test Cases:
In this phase the test cases generated and supplied by the testers are run. To run the test cases in MuJava there is an interface that contains a button "Run" to execute mutants against the test set supplied by tester and shows the results in the form of a mutation score of test set. The mutation score decides the accuracy and correctness of test set. the information of live and killed mutant is shown on the lower right hand side of the interface.

This is how the MuJava tool supports the Mutation Testing for JAVA.

VII. CONCLUSION

In this paper we have discussed the various techniques for performing the mutation testing that focus on both the test suites used and codes of the application to satisfy the need of the customer and software requesters. All the discussed methods have their own field of use. In the future work we will make the use of these techniques and will make some results out of that. The use of these techniques in real scenario is a big research oriented work so in future we will focus on the implementation of these techniques.
.

REFERENCES

[1]Marcio E. Delamaro, Marcos L. Chaim and Auri M. R. Vincenzi "*Twenty five years of research in structural and Mutation testing"*in 25th Brazilian Symposium on Software Engineering,978-0-7695-4603-2/11, IEEE 2011.

[2] Mike Papadakis and NicosMalevris "*An empirical evaluation of the first and second order mutation testing strategies*" in Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on,pp 90-99, IEEE 2010.

[3] Fevzi Belli, MutluBeyazit"*A Formal framework for mutation testing*" in proceeding of: Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, 2010.

[4]John A. Clark, Haitao and Robert M. Hierons "*Semantic Mutation Testing"*Science of Computer Programming, vol. 78 issue 4, pp. 345-363, 2013.

[5] S. Gossens, F. Belli, S. Beydeda, M. D. Cin, "*View Graphs for Analysis and Testing of Programs at Different Abstraction Levels*," in proceedings of the 9th International Symposium on High-Assurance Systems Engineering (HASE 2005), pp. 121- 13, IEEE CS Press, Oct. 2005.

[6] F. Belli, "*Finite-State Testing and Analysis of Graphical User Interfaces*," Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE CS Press, Nov. 2001, pp. 34-43.

[7] Munawar Hafiz "*Mutation testing tool for java*".

[8] Yu-Seung Ma, Jeff Offutt and Yong Rae Kwon "*MuJava: An Automated Class Mutation System*" Software Testing, Verification and Reliability,vol. 15(2): pp 97-133, June 2005.

[9] MarinosKintis,"*Isolating First Order Equivalent Mutants via Second Order Mutation*" in Proceeding ICST'12 ,Fifth International Conference on Software Testing, Verification and Validation, pp. 701-710, IEEE 2012.

[10]Sunwoo Kim John A. Clark John A. McDermid"Class Mutation: Mutation Testing for Object-Oriented Programs"In the Proceedings of the FMES 2000. October 2000.

[11] P. Chevalley and P. Th´evenod-Fosse, "A Mutation Analysis Tool for Java Programs," International Journal on Software Tools for Technology Transfer, vol. 5, no. 1, pp. 90–103, November 2002.

[12] Y.-S. Ma, "Object-Oriented Mutation Testing for Java," PhD Thesis, KAIST University in Korea, 2005

[13] Y.-S. Ma, Y.-R. Kwon, and A. J. Offutt," Inter-class Mutation Operators for Java," in Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02). Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, p. 352.

[14] B. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07), published with Proceedings of the 2ndTesting: Academic and Industrial Conference Practice and ResearchTechniques (TAIC PART'07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 193–202