IJITKM 0973-4414 Volume-10, Number-1 Jun-Dec 2016 pp. 40-49 **Impact Factor 4.2** available at www.csjournals.com

Test Case Generation: Problem and its solutions

Vikash Yadav

Research Scholar, Suresh Gyan Vihar University, Jaipur

Bright Keswani, Associate Professor, Dept. of Comp. Application, Suresh GyanVihar University, Jaipur

Abstract: Testing performs a very critical role for ensuring software quality. Software testing is the most commonly used technique for demonstrating that the software accomplishes its anticipated task. In the software testing one will check the actual output with the expected output. If both are equal then the behaviour of the software in normal otherwise the testing process needs revision. In software testing the software is executed with a set of test cases and the behaviour of the system for the test cases is evaluated to resolve if the system is performing as expected.

Keywords: Software Quality Assurance, Boundary Value, Data Flow Testing, Genetic Algorithms, Test Case Generation, etc.

1. Introduction

As software testing is the most time consuming phase in development, most of the problems associated with testing occur from one of the following causes:

(a) Failure to define testing objectives,

(b) Testing at the wrong phase in the cycle,

(c) Use of ineffective test techniques.

The main objective of testing is to deliver quality software. The cost of quality will have three components [11] namely the failure cost, appraisal cost & prevention cost. The success of testing in revealing errors depends significantly on the test cases. The process of testing includes choosing test data from the program's input domain, executing the program on these test data, and comparing the actual output with the expected output. The testing of the complete set of input would provide the complete depiction of the performance and functionality of the program. The complete set of input of a program is usually too large that it is impossible to test it for all. The exhaustive testers you will find but the exhaustive testing you will not. So the modus operandi of the reasonable software testing is to optimize the set of input by selecting relatively small subset, which will represent the entire input domain and the expected behaviour of the program on this set of input is then used to expect its behaviour in general. The test input should be chosen so that executing the program on this input set will expose every bit of errors. So any program which behaves accurately for small set of input will behave accurately for any set of input in the complete input set. An overwhelming majority of programs written nowadays handle data. Programming language paradigms exploit the concept of variables. Variables have been seen as the main areas where a program can be tested structurally. Numerous of variables in the program slice can be used together to calculate the values of other variables. Variables can receive their values from other sources like human interaction via a keyboard. This increased

level of complexity and can results errors in the programs. Value of variables may be altered in an unexpected way.

2. Approaches of testing

Different test case generation techniques have been developed for making the software testing successful. These include various black box and white box test case generation techniques. Some of them are described in following subsections.

2.1 Equivalence class partitioning

Before actually starting off the testing, it is good to have optimum test cases in place which not only cover all the required features but also are adequate to discover good quality bugs. There are many design techniques for writing test case. One of the most popular among them is Equivalence Class Partitioning.It is a Black-box (Specification Based) test case design technique with two primary goals [12] [19] [20]

1) To reduce the number of test cases to necessary minimum,

2) To select the right test cases to cover all possible scenarios.

The basic idea behind equivalence class partitioning is that the input for the program can be put into groups, and that the program should behave equivalently for each member of the group. Therefore, it will not be necessary to test each possible input, but only one or a few members of each equivalence class. As the exhaustive testing is next to impossible so the next natural approach is to divide the input domain into a set of equivalence classes. Now if any module or program work properly for any value in that class then it will work properly for all the other values lies in that equivalence class. Similarly one can design such equivalence classes for the entire input domain. So the number of test cases can be reduced by selecting one test case from each equivalence class. For example if one is checking a program to find whether a number is prime or not, then one can divide the entire input domain in to two equivalence classes one containing all the prime numbers and other containing all non-prime numbers. The prime numbers set of inputs are called valid inputs and non-prime set of inputs are called invalid inputs. For robust software one must also consider invalid inputs. Similarly one can further divide the classes in to other smaller classes to improve the testing process.

2.2 Boundary Value Analysis

Performance of any module at the border of each equivalence partition is more likely to be erroneous, so boundaries are an area where testing is likely to yield errors. The maximum and minimum values of a partition are its boundary values. A boundary value for a valid partition is a valid boundary value and the boundary of an invalid partition is an invalid boundary value. Tests cases can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is selected. Boundary value analysis can be applied at all test levels. It is comparatively easy to apply and its defect finding capability is high. This method of testing is often considered as an extension of equivalence partitioning. Suppose each input value has a defined range. The boundary value analysis can have six test cases. If any integer variable is having some minimum and maximum values, then there are six boundary values which fulfil the criteria of boundary value analysis. One has minimum-1, minimum, minimum+1 for lower limit and maximum-1, maximum, maximum+1 for upper limit. There are two strategies for combining the boundary values for the different variables in test cases. In the first case if one

has two variables X and Y then total 13 test cases (Xmin-1, Xmin, Xmin+1, Xmax-1, Xmax, Xmax+1, Ymin-1, Ymin, Ymin+1, Ymax-1, Ymax, Ymax+1, and one nominal value) are there. So in this case the total number of test cases becomes 6n+1. Secondly one can try all the possible combinations for the values for different variables. Now there are seven values for each variable so if there are n variables the total combinations will be 7n. There are certain limitations while optimizing test cases using BVA. Boundary value analysis works well when the software under test (SUT) is a function of several autonomous variables that represent bounded physical quantities. When these conditions are met BVA works well but when they are not there are lot of deficiencies in the results.

2.3 Cause-Effect Graphing

The major drawback with the above two techniques is that they consider each input individually. Both of these techniques do not focus on the combination of input that detects errors smartly rather these techniques focus on the conditions and classes of one input. A “Cause” in cause-effect graphing corresponds to an individual input condition that brings about an internal change in the system and an “Effect” represents an output condition. In this testing strategy first of all the input conditions called causes and their action called effect are identified for a module. Then a cause-effect graph is developed and transforms that graph into a decision table. Each column of a decision table represents a test case. If there are n different input conditions and any combination of the input conditions is valid then the total number of test cases comes out to be 2n. Cause-Effect graphing select combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. So after identifying the cause and effects, for each effect one can identify the causes that can produce that effect and how the condition has to be combined to make the effect true. The conditions are combined using the Boolean operators “AND”, “OR” and “NOT”. Cause Effect graphing technique generate high-yield test cases as well as gives the understanding of the functionality of the system. There are lots of techniques available for reducing the number of test cases generated by appropriate traversing of the graph [19] [20].

2.4Control-Flow Testing

Control-flow testingis one of the structural testing techniques that use the program’s control flow as a model. Control-flow testing applies to almost all software and is effective for most software. It is a fundamental testing technique. Its applicability is mostly to relatively small programs or segments of larger programs. Control-flow testing techniques are based on judiciously selecting a set of test paths through the program. The set of paths chosen is used to achieve a certain measure of testing thoroughness for examplepick enough paths to assure that every source statement is executed as least once. Control-flow testing is most applicable to new software for unit testing. Control-flow bugs are not as common as they used to be because structured programming and object oriented languages minimize them.

The postulations of Control-flow testing are

1) Specifications are correct;

2) Data is defined and accessed properly;

3) There are no bugs other than those that affect control flow.

Fundamental Path Selection Criteria are:

1. Ensure that every instruction in the routine has been exercised at least once;
2. Every decision has been taken in each possible direction at least once;
3. An adequate number of paths to achieve coverage;
4. Selection of short, functionally sensible paths;
5. Minimizing the number of changes from path to path. Preferably only one decision changing at a time;
6. Favour more but simpler paths over fewer and complicated paths;
7. Outline of Control Flow Based Testing;
8. Inputs to the test generation process
    a. Source code
    b. Path selection criteria: statement, branch, etc.
9. Generation of control flow graph (CFG)
    a. A CFG is a graphical representation of a program unit.
    b. Compilers are modified to produce CFGs. (You can draw one by hand.)
10. Selection of paths
    a. Enough entry/exit paths are selected to satisfy path selection criteria.
11. Generation of test input data
    a. Two kinds of paths
        i. Executable path: There exists input so that the path is executed.
        ii. Infeasible path: There is no input to execute the path.
    b. Solve the path conditions to produce test input for each path.

The control flow graph is a graphical representation of a program's control structure. Flow graphs consist of three primitives,

1) Adecision is a program point at which the control can diverge. e.g. if and case statements).

2) A junction is a program point where the control flow can merge. (*e.g.,* end if, end loop, goto label)

3) A process block is a sequence of program statements uninterrupted by either decisions or junction's i.e.straight-line code.

A process has one entry and one exit.

A program does not jump into or out of a process.

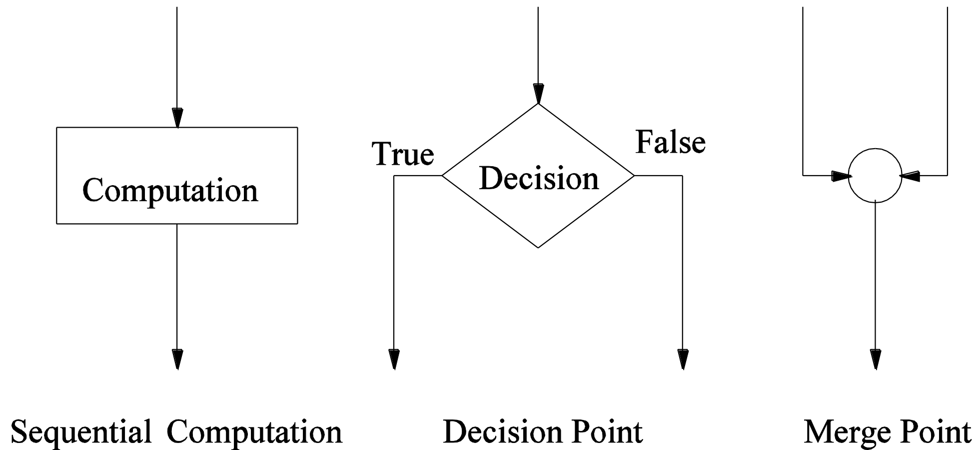

Figure 1: Symbols in a control flow graph

2.5 Data Flow Testing

The idea of data-flow testing permits the tester to inspect variables throughout the program to discover errors. The data-flow testing is a form of structural testing that is a variant on path testing. It focuses on the dentition and usage of variables, rather than the structure of the program. Data-flow testing permits the tester to chart the changing values of variables within the program. It does this by utilising the idea of a program graph. So data-flow testing is closely related to path testing but the paths in the program are selected on variables. Data-flow testing looks at the life-cycle of a particular piece of a variable in an application. By looking for patterns of data usage, risky areas of code can be found and more test cases can be applied on it.

There are four ways data can be used

1) Defined,

2) Predicate use (pu),

3) Calculation use (cu),

4) killed.

Some patterns using a piece of data in a predicate logic after it has been killed show an anomaly in the code, and therefore the possibility of a bug [12] [20].

2.6 Mutation testing

Mutation testing has been around since the late 1970s but is rarely used outside academia. Executing a huge number of mutants and finding equivalent mutants has been too expensive for practical use. Mutation testing measures how "good" our tests are by inserting faults into the program under test. Each fault generates a new program, a mutant that is slightly different from the original software module. The idea is that the tests are adequate if they detect all mutants.

Mutation testing is a coverage criterion that has its roots in the very definition of reliable test sets. This is what makes it fundamentally different from most other criteria. Fault-based testing is the way in which some "marked" bugs loose in the code and try to catch them. If one catches them all, then "net" probably caught many of the other, fishier, fish. The unknown bugs, that is.

One of the fault-based testing strategies is mutation testing. There are many variations of mutation testing such as weak mutation [10], interface mutation [5] and specification-based mutation testing [19].

3. Optimization of test cases

Software testing is the most commonly used technique for demonstrating that the software accomplishes its anticipated task. The process of testing includes choosing test data from the program's input domain, executing the program on these test data, and comparing the actual output with the expected output. The testing of the complete set of input would provide the complete depiction of the performance and functionality of the program. The complete set of input of a program is usually too large that it is impossible to test it for all. The exhaustive testers you will find but the exhaustive testing you will not. So the modus operandi of the reasonable software testing is to optimize the set of input by selecting relatively small subset, which will represent the entire input domain and the expected behaviour of the program on this set of input is then used to expect its behaviour in general. The test input should be chosen so that executing the program on this input set will expose every bit of errors. So any program which behaves accurately for small set of input will behave accurately for any set of input in the complete input set.

An overwhelming majority of programs written nowadays handle data. Programming language paradigms exploit the concept of variables. Variables have been seen as the main areas where a program can be tested structurally. Numerous of variables in the program slice can be used together to calculate the values of other variables. Variables can receive their values from other sources like human interaction via a keyboard. This increased level of complexity and can results errors in the programs. Value of variables may be altered in an unexpected way.

4. Genetic Algorithms

A genetic algorithm is a form of evolution that occurs on a computer. They are search methods that can be used for both solving problems and modeling evolutionary systems. The Darwinian theory of evolution depicts biological systems as the product of the ongoing process of natural selection. Likewise, genetic algorithms allow engineers to use a computer to evolve solutions over time, instead of designing them by hand. Forrest has first explained the working of genetic algorithm and then discussed its application as problem solver and for making models [7]. With various mapping techniques and an appropriate measure of fitness, a genetic algorithm can be tailored to evolve a solution for many types of problems including optimization of a function or determination of the proper order of a sequence. She discussed the use of genetic algorithms in modeling ecological systems, immune systems and social systems. Mathematical analysis of genetic algorithms using the Holland’s schema theorem and building block hypothesis has also been discussed in the article.

David Goldberg explained the genetic algorithms and evolutionary algorithms as family of computational methods in Darwinian Evolution. Genetic algorithms are search procedures based on natural selection and genetics. A simple genetic algorithm consists of selection, crossover and mutation. Selection is survival of fittest within the genetic algorithm. The key notion of selection is to give preference to better individuals. The design methodology of genetic algorithm relies heavily on Holland’s notion of schemata and building blocks. He stressed that genetic algorithms are best suited for wide range of applications because they can solve hard problems quickly and

reliably. Genetic algorithms are extensible, easy to hybridize and easy to interface to existing simulations and models [8].

Goldberg also reviewed the elements of genetic algorithms and described the mechanics of genetic algorithms in his work [9]. He developed the fundamental intuition of genetic algorithms or innovation intuition. According to him,

Selection + Mutation = Continual Improvement

Selection + Recombination = Innovation

He also discussed the technical lessons of genetic algorithm design. The primary idea of selecto-recombinative genetic algorithm theory is that genetic algorithms work through a mechanism of quasi-decomposition and recomposition. Genetic algorithms implicitly identify building blocks or sub-assemblies of good solutions and recombine different subassemblies to form very high performance solutions. They suggested that research into genetic algorithms is profoundly changing. Genetic algorithm research helps us identify some of the different facets of innovation quantitatively. Genetic algorithm teaches us to respect the generation of an outstanding individual. Genetic algorithm research teaches us that creativity essentially makes hard problems easier by either directly or indirectly making the building blocks necessary to solve the problem more accessible to the search.

Mitchell *et al.* analyzed two algorithms – RMHC and IGA to identify general principles of when and how a genetic algorithm will outperform hill climbing [16]. Experimental analysis was carried on Royal Road landscape. RHMC was analyzed with respect to R1. Genetic algorithm proved to be fast because of implicit parallelism. IGA also perfectly implements implicit parallelism. Expected time for IGA is on the order of 2K log N and for RHMC is on the order of 2K N log N. RHMC is slower than IGA. Analysis was further carried on modified Royal road landscape R4 to understand how genetic algorithm works in general and where it will be more useful.

Melanie Mitchell has explained the term Biological computation and its relation to computational biology and biologically inspired computing [17]. She then compared Biological computing with Traditional computing. In case of traditional computers, the processing of information is centralized and performed by the CPU. Traditional Computing require synchronization in many aspects of their processing. Traditional computing systems require components to be reliable with very low error probabilities. In Biological computing, information processing is massively parallel, stochastic, inexact and ongoing with no clean notion of a mapping between inputs and outputs. Biological systems operate with asynchronous components. Biological systems operate with unreliable components that are subject to frequent failures. In traditional computer science, universal computation and programmability are fundamental whereas relevance of these concepts for biological computing is unclear. Mitchell also analyzed the question –*"Is computing a Natural Science?"*

Korf and Reid analyzed the asymptotic time complexity of admissible heuristic search algorithms such as A*, IDA* and depth first branch and bound. Time complexity of these algorithms depends primarily on the quality of the heuristic function [13]. Korf and Reid characterized heuristic function simply by the distribution of heuristic values in the problem space. Experimental analysis was carried out on Rubik's cube, Eight Puzzle, Fifteen Puzzle. Analysis showed that the asymptotic heuristic branching factor is the same as the brute force

branching factor and the effect of a heuristic function is to reduce the effective depth of search rather than the effective branching factor. Asymptotic analysis was presented for fixed size problem as the solution length grows large and provided excellent predictions at typical solution depths.

Meng *et al.* studied various encoding techniques in genetic algorithms and present sufficient convergence condition on genetic encoding in genetic algorithms [14]. They identified new categories of code like uniform code, bias code, trisector code and symmetric code and applied them on classical genetic techniques. Simulation results showed that genetic algorithms with specific codes can find solutions with better quality in shorter time than classical genetic algorithms. They also concluded that there is significant influence of encoding techniques on genetic algorithm's performance in solving problems with big algorithm complexity.

Evolvable hardware is upcoming application area, thereby; synthesis of analog and digital electronic circuits through evolving algorithms is gaining attention in current researches. In circuit synthesis, chromosomes represent a circuit and each of its genes describes the component of the circuit. Mesquita *et al.* proposed the adjacency matrix representation for chromosomes in evolving circuits. Adjacency matrix representation reduces the generation of anomalous circuits unlike earlier Incidence matrix, thus, increasing the efficiency of overall process [15]. They tested their proposed encoding for variable size chromosomes – their concatenation and cascading. Adjacency matrix assumed a graph with no parallel branches and thus prevents explicit representation of individual circuit elements.

A large number of scheduling problems exist in domain of optimization problems. A schedule is constructed such that some measure is reduced. Commonly scheduling problems are modeled as a graph. Reviewing the pros and cons of earlier representation schemes for scheduling problems, Fenton and Walsh stated that repeating permutation representation has high volume of redundancy but it is useful and robust [6]. They introduced variety of genetic operators for repeating permutation representation like GMOX, GPX, GUX, PPX, PBM, SBM, and OBM. They tested these operators using GALIB. In all trials, GMOX outperformed other operators. Morphogenic computation yielded better results and improved evolvability of genetic algorithm.

Genetic algorithm with chromosome differentiation (GACD): Nature generally differentiates the individuals in the species into more than one class. The prevalence of differentiation indicates an associated advantage which appears to be in terms of cooperation between two dissimilar individuals who can at the same time specialize in the own fields [21]. GACD incorporates chromosome differentiation for evolutionary process. Chromosomes are distinguished into two categories of population over the generations based on the value contained in the two class bits. These are initially generated based on maximum hamming distance between them. Crossover (mating) is allowed only between individuals belonging to these categories [1]. Theoretical analysis shows that the basic tenet of genetic algorithms holds for GACD as well; above average, short, low order schema will receive increasing number of trials in subsequent generations. It is proved that in many cases the lower bound of the number of instances of a schema h sampled by GACD is greater than or equal to that of CGA. Because of this, GACD is better able to exploit the information gained so far. Again, initializing the M and F populations in such a way so as to maximize the hamming distance between them, and allowing mating between individuals from these two dissimilar populations, enhances the exploration capability of GACD. Therefore, GACD appears to strike a better balance between exploration and exploitation, which is crucial

for any adaptive optimization technique, thereby giving it an edge over the conventional Genetic Algorithm.

In 1993, De Jong and Sarma presented additional empirical evidence and suggested alternative deletion methods to reduce the variance [3] [4]. Cobb & Grefenstette compared three different strategies and modified the standard genetic algorithm in order to make it more applicable to rapidly changing environments [2]. A partial hyper mutation step was introduced after mutation which replaced a percentage of population by randomly generated individuals. The percentage replaced was called replacement rate. In order to measure the effect of replacement rate, 23 modified genetic algorithms on non-stationary test functions were considered with varying percentage of population. Experiments showed that 10% and 30% random replacement gave better tracking performance. 50 % replacement showed too much random exploration.

DeJong was the first to evaluate empirically the performance of genetic algorithms with overlapping populations. DeJong also stated the concept of crowding that follows the simple genetic algorithm except that only a fraction of population reproduces and dies each generation [3]. He introduced the generation gap G as a parameter to genetic algorithm where a percentage of population is chosen via fitness proportionate selection to undergo crossover and mutation and G x n individuals from population are chosen to die. He found that at low values of G, the algorithm had severe losses of alleles, also known as genetic drift, and resulted in poor search performance.

5. Conclusion

Software testing is very crucial part of software development. For testing we need some good quality inputs for the software and when we run software with these inputs we have to check the behavior of software. If these inputs are not good, testing may not be effective. So many test cases are required for software, which can be generated by Genetic Algorithm. So, test cases generation can be treated as an optimization problem and we can use Genetic Algorithm to solve it.

References

1. Bandyopadhyay, S., S.K.Pal and U.Maulik, (1998) Incorporating chromosome differentiation in genetic algorithms, *Information Sciences*, 104, pp 293-319.
2. Cobb H.G. and J.J. Grefenstette, (1993) Genetic algorithms for tracking changing environments, In S.Forrest (Ed.) *Proceedings of the Fifth International Conference on Genetic Algorithms,* San Mateo CA Morgan Kaufmann, pp 523-530.
3. De Jong, K.A., (1975) An Analysis of the behavior of a class of genetic adaptive systems. (Doctoral dissertation, University of Michigan), 36(10), 5140B (University Microfilms No. 76-9381).
4. De Jong, K.A. and J.Sarma (1993) Generation Gaps revisited. In D.L.Whitley (ed.) *Foundations of Genetic Algorithms 2*, Morgan Kaufmann, pp 19-28.

5. Delamaro M. E., J. C. Maldonado & Mathur A. P.(1996): Integration Testing Using Interface Mutation, Proceedings of the Seventh International Symposium of Software Reliability Engineering (ISSRE'96), White Plains, NY, pp.112–121.
6. Fenton, P. and P.Walsh (2005) Improving the performance of the repeating permutation representation using morphogenic computation and generalised modified order crossover, In *Proceedings of Congress on Evolutionary Computation 2005*, pp 1372-1379.
7. Forrest,S., (1993) Genetic Algorithms: Principles of Natural Selection Applied to Computation, *Science*, Vol.261, No.1, pp 872-878.
8. Goldberg, D.E., (1994) Genetic and Evolutionary Algorithms Come of Age, *Communications of the ACM*, Vol.37, No. 3, pp 113.
9. Goldberg, D.E., (2000) The design of innovation: Lessons from genetic algorithms, lessons for the real world. Technological Forecasting and Social Change.
10. Howden W. E.(1982): Weak mutation testing and completeness of test sets, IEEE Trans. on Softw. Eng., 8(4), pp371–379.
11. Humphrey W. S.(1997): Introduction to the Personal S/W Process, Addison Wesley Longman Inc., 1997.
12. Jorgensen P. C. (2001): "*Software Testing: A Craftsman's Approach*". CRC Press, 2nd edition.
13. Korf, R.E., M. Reid, (1998) Complexity Analysis of Admissible Heuristic Search, In *Proceedings of the National Conference on Artificial Intelligence (AAAI-98),* Madison, WI, pp 305-310.
14. Meng, Q. C., T.J.Feng, Z.Chen, C.J.Zhou and J.H. Bo, (1999) Genetic algorithms encoding study and a sufficient convergence condition of GAs, In *Proceedings of 1999 IEEE International Conference on Systems, Man, and Cybernetics*, Tokyo, Japan, Vol. 1, pp. 649-652.
15. Mesquita, A., F.Salazarand and P.Canazio, (2002) Chromosome Representation through Adjacency Matrix in Evolutionary Circuit Synthesis. NASA/Conference on Evolvable Hardware, ISBN 0769517188,pp 102-112.
16. Mitchell,M., J.H.Holland and Stephanie Forrest (1994) When will a Genetic Algorithm Outperform Hill Climbing?, In J.D.Cowan. G.Tesauro and J.Alspector (Eds.) *Advances in Neural Information Processing Systems,* 6, San Mateo, CA, Morgan Kaufmann.
17. Mitchell, M., (2011) What is Computation? – Biological Computation, *Ubiquity,* an ACM PUBLICATION.
18. Sonia Bhargava, Bright Keswani, "Generic ways to improve SQA by meta-methodology for developing software projects", International Journal of Engineering Research and Applications, Vol. 3, Issue 4, pp 927-932, July 2013.
19. Murnane T. & Reed K.(2001): On the Effectiveness of Mutation Analysis as a Black Box Testing Technique, 13th Australian Software Engineering Conference (ASWEC'01), Canberra, Australia p0012.
20. Myers, G.J. (1979): *The Art of Software Testing*, John Wiley & Sons, Inc., New York.
21. Sivaraj, R. and T. Ravichandran, (2011) A review of selection methods in Genetic Algorithms, *International Journal of Engineering Science and Technology,* Vol.3, No.5, pp 3792-3797.